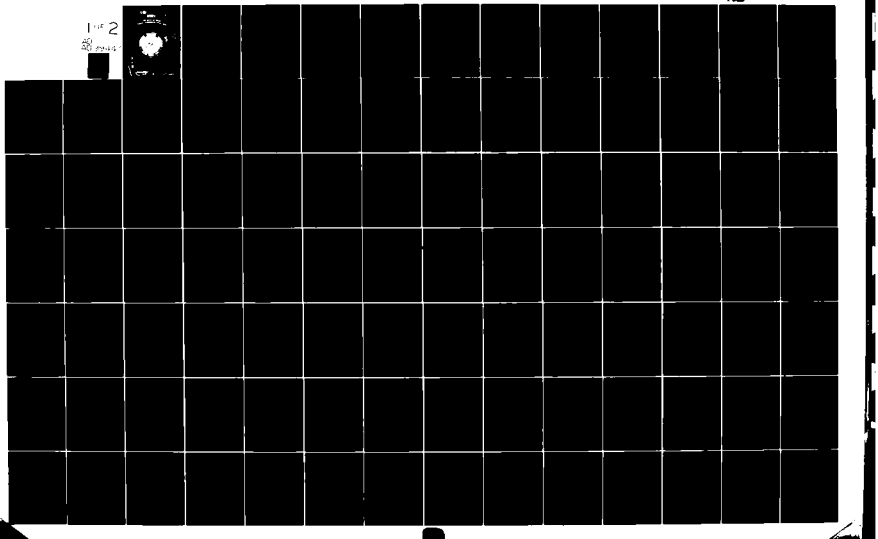
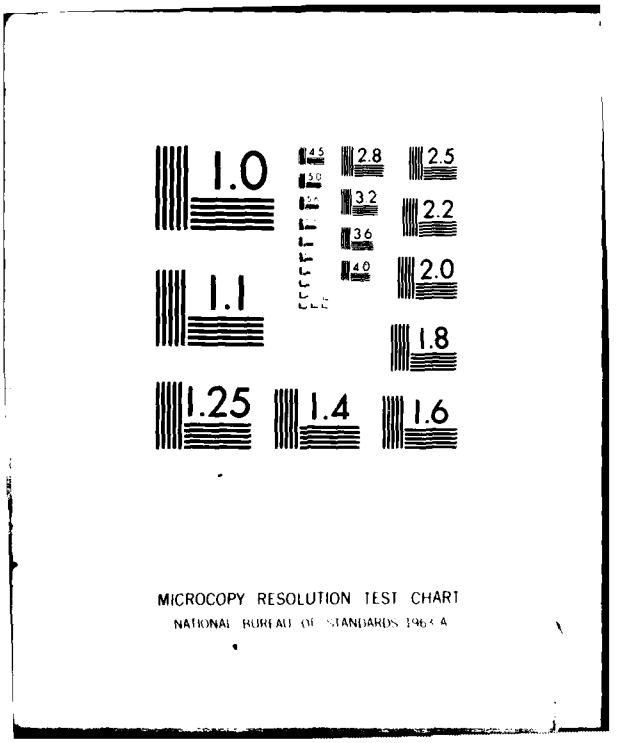
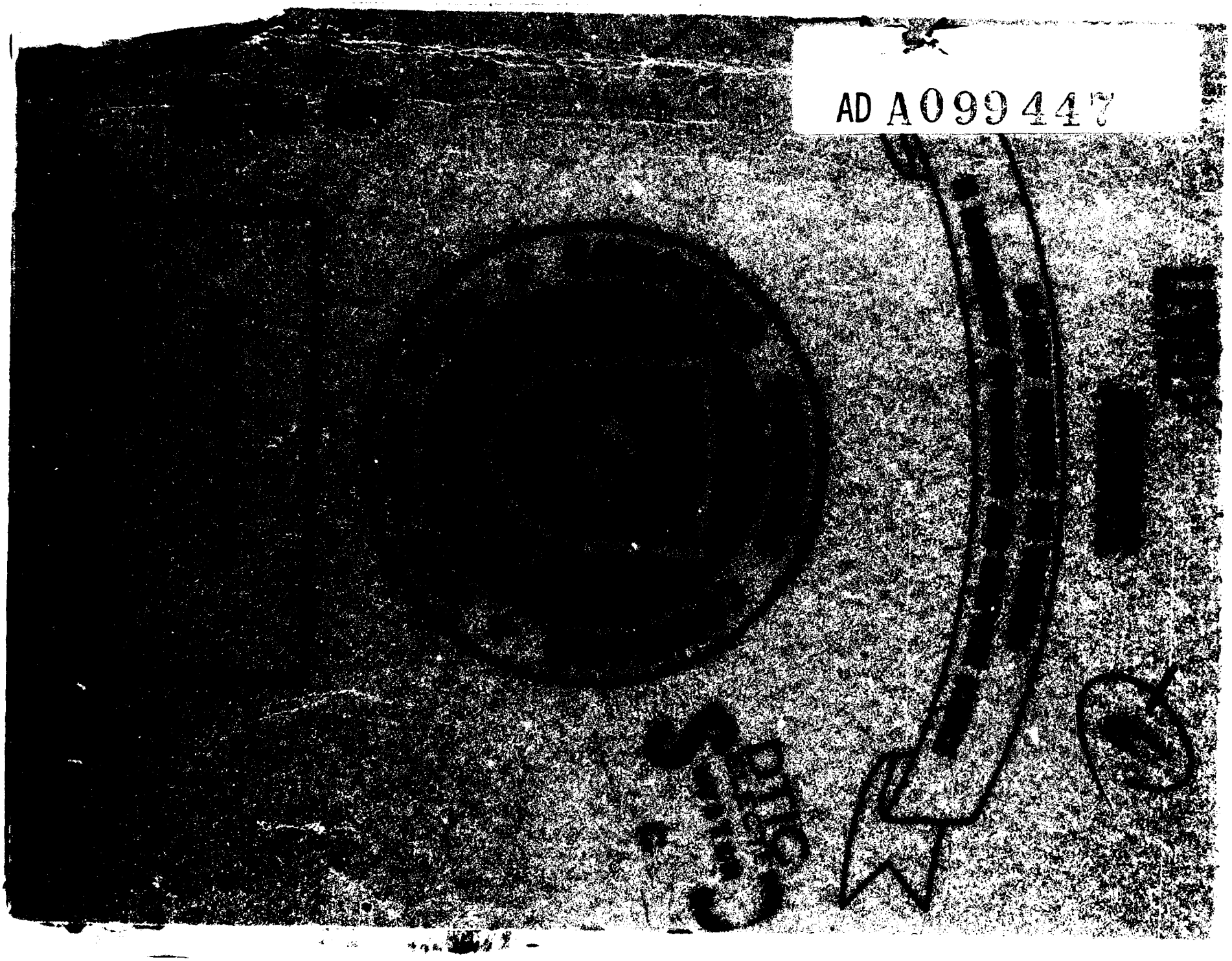
AD-A099 447 ARMY INST FOR RESEARCH IN MANAGEMENT INFORMATION AND --ETC F/G 9/2
ADA - A SUITABLE REPLACEMENT FOR COBOL?(U)
FEB 81 J S DAVIS

UNCLASSIFIED NL




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963-A



DISTRIBUTION STATEMENT

Approved for public release. Distribution unlimited.

The views, opinions, and findings contained in this report are those of the author and should not be construed as official Department of the Army position.

This technical report has been reviewed and approved.

Clarence Giese, Director
U.S. Army Institute for Research
In Management Information
and Computer Science, U.S.
Army Computer Systems Command

John R. Mitchell
Chief, Computer Science
Division, U.S. Army
Institute for Research
and Computer Science, U.S.
Army Computer Systems
Command

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (6) 2. GOVT ACCESSION NO. AD-A099447	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Ada - A Suitable Replacement For COBOL?	5. TYPE OF REPORT & PERIOD COVERED (9) Final rept.
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) MAJ John S. Davis	8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS US Army Institute for Research in Management Information and Computer Science (ACSC-AT) 115 O'Keefe Bldg, G.I.T., Atlanta, GA 30332	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS (11)	12. REPORT DATE 24 Feb 1981
	13. NUMBER OF PAGES 98
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) (Same as 9) (12) 101	15. SECURITY CLASS. (of this report) Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Programming language, High Order Language, Standardization.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The new Department of Defense standard programming language for embedded computer systems, Ada, is evaluated as a replacement for COBOL. Ada appears superior to COBOL in facilitating good software development and maintenance practices. Yet Ada is more difficult to learn and does not provide as many convenient built in features for data formatting and input/output. Adopting Ada may reduce total life cycle cost, but converting from COBOL to Ada is not recommended for the near future.

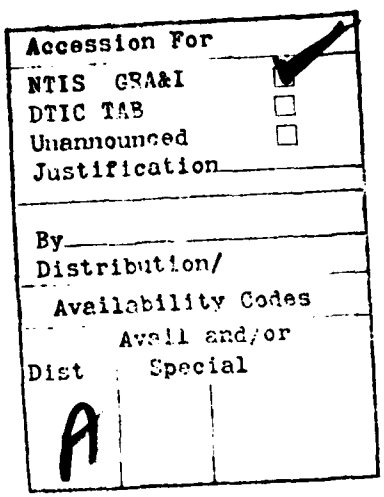
DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

Unclassified 394679

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

24 February 1981

Ada -- A Suitable Replacement for COBOL?



Major John S. Davis
Army Institute for Research in
Management Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

Table of Contents

Research Objective

The objective of this project is to determine the technical feasibility of replacing COBOL with Ada within the Department of Defense (DoD), and thus establishing Ada as the standard programming language for management information systems (MIS). Motivation for this study is the growing support of Ada, both in government and the private sector, for a wide variety of applications, not just the originally intended embedded computer software. There is a need to rationally evaluate Ada for MIS. This will provide some basis for MIS developers to welcome or reject future proposals to extend the scope of the DoD Ada standardization decrees.

Rationale for Ada

> **Let us go down, and there confound their language,**
> **that they may not understand one another's speech.**
> **(the book of Genesis)**

The Army, Navy and Air Force have independently developed or used an assortment of programming languages, many of which are used by only one service [GLAS79]. This proliferation hinders sharing of products and services. Also a problem is the widespread use of assembler language [MART79]. Assembler code is difficult (and therefore costly) to understand, debug or modify. The difficulty to change assembler software is significant because military requirements and environments are particularly subject to change, and necessary alterations to

software can consume more resources than did the original development. Assembler code is difficult to test, since algorithms are usually obscured in a maze of microscopic detail. Therefore the testing process is costly, and critical errors may survive and plague the delivered system. Since assembler languages are usually unique for each different machine, software is not portable. It is restricted in application. Thus use of assembler contributes to software proliferation, and the military is vulnerable to support problems which could occur if the sole source of a machine goes out of business or discontinues a product line. The manufacturer may be the only source of assembler language support.

About half of Department of Defense software costs are attributed to embedded computers, that is, computers which are themselves components of a weapons system [BUXT80]. This software often has distinctive characteristics: the ability to continue to operate after hardware or software faults, connection with unusual peripheral equipment, monitoring of sensors, control of special displays, and the need to respond to real time events [BUXT80]. Assembler seems appropriate in many applications where needed features do not exist in high order languages (HOL), and therefore compiler generated code is inefficient.

In recent years hardware costs have been decreasing as software costs have risen, and the run time inefficiencies of HOL have become more tolerable. Particularly attractive to Department of Defense is the potential increase in programmer productivity. Assuming it takes about the same effort to produce one line of assembler code as one line of HOL code, and if one line of HOL code produces 10 lines of

assembler code, great savings are possible during development and during implementation of system changes [MART79].

Department of Defense has taken a number of strong measures to combat software proliferation. Realizing the increasing benefits of HOL programming, to achieve a short term impact they issued Directive 5000.29, which required use of HOL unless it could be "demonstrated that none of the approved high order languages are cost effective or technically practical over the system life cycle" [GLAS79]. Shortly afterward, Directive 5000.31 specified seven existing high order languages as the only ones permitted for Department of Defense systems: FORTRAN and COBOL (widely used); TACPOL (Army); CMS-2 and SSPL (Navy); JOVIAL, J3 and J-73 (Air Force) [MART79]. In 1975 a High Order Language Working Group (HOLWG) was commissioned to develop a common HOL for Department of Defense [WHIT79].

The HOLWG was not christened impulsively. A 1977 study by MITRE Corporation, based on extremely conservative assumptions, estimated that a common language would save $110 million to $900 million during 1983 - 1999 [MART79]. Studies by the Defense Advanced Research Projects Agency and by Decisions and Designs, Inc., based on more realistic assumptions, predicted savings of $12 to $24 billion [MAR79]. The initial HOLWG objective was to determine requirements for a common language. A series of draft requirements documents were developed and circulated among military departments, government agencies, industry and academic institutions. As reactions to each draft were evaluated, the product became more solid. The names of the papers reflect this solidarity: STRAWMAN (1975), WOODENMAN, TINMAN,

IRONMAN , STEELMAN (1978) [GLAS79]. The proposed new language was named Ada, after Ada Augusta -- the world's first programmer [ICHB79]. She worked for Charles Babbage during his development of the Difference Engine and Analytical Engine circa 1829 [GLAS79].

After evaluating existing languages, the HOLWG found none adequate to handle the requirements, but three were selected as candidate base languages: Pascal, PL/1 and ALGOL 68. They did conclude that it would be practical and entirely satisfactory to develop a derivative of an existing language rather than an entirely new one. In 1977 four contractors were selected to design the Ada language using any one of the base languages. All chose Pascal. In April 1979, the design of Cii - Honeywell Bull was declared the winner [GLAS79]. The Department of Defense has since sponsored Ada training sessions for interested industry representatives, and the HOLWG has considered their feedback in further refining Ada.

Certain to add momentum to the Ada program is the Army's award of a contract in 1980 for development of an Ada compiler for the VAX 11/780. The compiler should be completed in 1982, tested and certified by 1983 [LEI80B]. The HOLWG plans development of an Ada Programming Support Environment, a set of integrated software tools for Ada software development. When completed, the Army will install this environment on a VAX to establish a software development facility [11]. The target date for the complete facility is 1983 [LEI80B].

Certain Ada features are directly related to the typical military requirements described earlier:

- Facility for parallel tasks (initiation, termination, rendezvous) for real time systems [ICH79B].

- Exception handling, for establishing capability to survive hardware or software errors [ICH79B].

Other features facilitate system development and portability [ICHB79]:

- Separately compilable program units.

- User defined abstractions whose implementation details may be "hidden".

Military Computer Family

Ada is oriented towards embedded computers. Yet DoD may in the future pursue even greater standardization by establishing Ada as the common HOL for all applications, including MIS. A related Army project, Military Computer Family (MCF) may give added impetus to this idea. The MCF proposes to attack hardware proliferation by fielding an instruction-set-compatible family of battlefield computers, designed to be efficient target machines for Ada compilers. An in depth study of future Army requirements shows conclusively that two computer models, a micro and a mini, will satisfy virtually all battlefield applications in the 1980's [COMP80]. Accepting the study results, the MCF project proposes development of a micro and a "super mini" model using 1984 technology for 1986 delivery [LEI80B]. Hardware specifications are based on forecasts of the 1984 state of the art in order to postpone the technology lock-in as long as possible.

The microcomputer, dubbed the AN/UYK-41V1, is to satisfy the following constraints:

- Size: Contained on a 6" by 9" card (not including power supply).

- Weight: Less than 12 ounces.

- Speed: 500,000 instructions per second.

- Store: 128K bytes.

- Cost: $5,000 in 1980 dollars.

- Reliability: 100,000 hours mean time between failures.

- Power: Less than 5 watts.

Though not required, it is quite possible that this micro might be able to perform as a component of its big brother, the "super-mini" AN/UYK-41V2.

The V2 specifications are listed below:

- Size: Contained in an air transport rack, 7" tall, 10" wide and 13.5" deep (.5 cubic feet).

- Speed: 3,000,000 instructions per second.

- Store: 2 Megabytes.

- Cost: $75,000 in 1980 dollars.

- Reliability: 10,000 hours mean time between failures.

- Power: 100 watts.

The size of the V1 and V2 computers allows a wide range of application among battlefield systems. The larger V2 is about the same size as the ubiquitous Army FM radio (VRC-12, VRC-46) commonly installed in jeeps, armored personnel carriers and tanks. The V1 is even smaller than the standard Army manpack radio (PRC-77).

The Army will select up to four contractors to completely design the MCF models by 1983. Recall that the inner workings of the computer box are largely at the discretion of the manufacturer, so each

contractor may develop an independent design. Producers of the two best designs will be given the go-ahead to develop prototypes by 1985. After extensive testing, the winner will be selected for full scale production beginning in 1985. First production models should appear in 1986.

In order to transition smoothly to the MCF, the Army will develop a "software MCF" by 1983: an Ada compiler, MCF simulator and tactical operating system on a commercial host computer. Therefore software targeted for MCF models can be developed and tested as the computers are being built [LEIB80]. Further information on the MCF project and architecture of the MCF machines is contained in an earlier report [DAVI80].

Implications of Ada/MCF for MIS

The characteristics of the MCF computers make them front runner candidates for battlefield and other management information system hardware. For example the MCF mini will be smaller and more powerful than the van mounted DAS3 minicomputer being fielded in 1980 to support logistical MIS for maintenance units. MCF computers are logical replacements for the DAS3 and for the aging IBM 360/30 machines serving Army divisions. Many Army activities are formulating plans for increased automation of the battlefield, including new applications of MIS using portable or mobile computers.

A recent Army policy memorandum requires that all future automated battlefield systems use MCF hardware and Ada software [ARMY80]. At this time it is not clear whether the policy applies to

management information systems which operate in whole or in part in the battlefield environment. I see the handwriting on the wall. I expect Ada to be used for future battlefield MIS.

Since embedded computers (and hence MCF) will be purchased in record numbers during the next ten years, the inevitably rich, Ada-oriented base of software development tools and application programs will increase the attractiveness of using Ada for MIS. Future tactical MIS will interface with other major components of a battlefield automated system. A common language and support environment will contribute to an integrated logistics concept with the potential for considerable cost reduction.

What is a Management Information System?

Management information Systems (MIS) are designed to "aid management in organizational planning, operation and development" [ENCY76]. There appears to be no widely accepted definition beyond this general description.

Managers are, the name MIS implies, an important class of users of the system. The information system helps them plan, make decisions and control their organization. Features often found in MIS are [ENCY76]:

- a database representing the organization and its environment
- simulation capability, based on the database or a "corporate model" derived from it
- decision support tools
- information summaries and analyses
- an information retrieval system.

The term MIS in this paper refers to the broadest possible

context; emphasis is on "information system" rather than on "management." I believe this describes a meaningful class of systems which have evolved from the original "ADP" systems.

When computers were first introduced, organizations employed them as labor saving devices. Clerical functions were automated to reduce administrative labor and cost. Routine, recurring functions were the principal targets for automation (inventory, payroll, etc.). Usually the automated system was a mirror image of the corresponding manual system. Data was stored in master files, usually magnetic tape, and processing consisted of having transactions update the master files.

Many MIS currently in use resemble the old fashioned ADP systems: they are based on sequential files and periodic processes which operate on them. Summary reports do help managers make decisions, but such MIS support the activities of personnel throughout the organization. A stock control system may interface with salesmen, stock clerks, finance clerks and others -- it is not necessarily just a management tool.

The trend in modern MIS, made possible by more prevalent random access devices, is toward integrated database management systems. This trend reflects the need to correlate information on all aspects of an organization. Formerly data was fragmented among the master files of the various MIS. It could not conveniently be accessed or processed other than through the individual MIS. Also reflected in this trend is the requirement to respond to changes in managers' needs. The summary reports of the old style MIS are frozen in content and format. Revising them requires a time consuming and expensive reprogramming effort.

Perhaps MIS, as used here, can be most concisely defined as "systems which aid management and others in organizational planning and operation."

Characteristics of MIS include:

- large volume of data (input, master files and output)
- relatively simple transactions on the data
- many routine manual processes are emulated or supported
- large, complex systems having numerous processes and interfaces with many different users or other MIS.

The COBOL Realm

In this report COBOL refers to the language defined in "American National Standard Programming Language COBOL," ANSI X3.23 0 1974 [ANSI74]. Though a new version may be released in 1980, the analysis in this paper remains valid. Initially defined in 1959, COBOL is a full twenty years older than Ada.

COBOL is designed to facilitate handling large volumes of data, such that input/output is one of the most important aspects of a typical program. Use of large files on external mass storage devices is endemic. This is "business data processing," an application realm where perhaps the most characteristic function is to process a sequence of records, performing rudimentary computations on each and putting the results into another set of records. Management Information Systems (MIS) have this same flavor, sometimes with the addition of statistical or decision making packages.

COBOL's age and popularity give it an enormous headstart over the newcomer Ada. There are more programs written in COBOL than any other language [SHAW78].

Application Environment

The COBOL application environment chosen for this analysis is that of the U.S. Army Computer Systems Command, which develops standard Army management information systems for Army units and installations worldwide. Systems developed include personnel, logistics and financial management systems bearing strong resemblance to their counterparts in the business community. Thus, there is some justification in the assumption that a representative environment exists for COBOL programming in general.

Decision Factors

The decision whether to adopt a new standard language for a class of applications is impacted by many considerations, which I group into two categories:

- "steady state" factors (how suitable is the language for coding new programs?)
- transition factors (how costly is the conversion to a new language?)

The latter category includes retraining of programmers and conversion of existing software, unless it is to be maintained in the original language for the duration of the life cycle. I suspect that though they are significant and may be paramount, transition costs are

similar for adoption of any new language. Retraining of programmers will of course be less if the new language does not require a major reorientation of the programmer's approach.

Since the transition is the first issue to be conquered, the Computer Systems Command commander is justifiably concerned about the difficulty of a conversion to Ada. An analysis of this problem including alternative approaches, estimation of dollar costs and estimation of impact on service to the user community, is essential to the making of an intelligent decision on the adoption of a new language. Yet in the long run, steady state costs (or savings) will overshadow start-up costs.

I concentrate here on examining the steady state, technical issues, and defer until later a study of the transition problem. I do take somewhat of a transition oriented approach in that I attempt to view Ada through the eyes of a "COBOL-minded" programmer. Niklaus Wirth [WIRT74] might object to this approach, since it seems to be concerned with giving the COBOL programmer what he thinks he wants rather than what he needs. Wirth would support Ada for MIS if it proved to be a language with a sound (though initially foreign) set of tools for solving the problems confronting today's COBOL programmer.

Research Approach

I select what I believe to be some of the key features of COBOL and compare what Ada has to offer on a feature by feature basis. In the discussion of each feature I explain why the feature is or is not important. The Ada solutions to COBOL features are in many cases an

emulation of COBOL constructs, intended to be rather easy for a newly converted COBOL programmer to grasp. There is no requirement for a new language to provide parallel capabilities or to mimic constructs of the language it is to replace. Probably most COBOL programmers, after mastering Ada, would employ a new programming style in the spirit of the new language. Criticism may be levelled at the idea of considering COBOL at all; perhaps the characteristics of business data processing should be the prime criterea, not the COBOL language. I proceed without resolving that question, but make the assumption that there is some merit in comparing against the time proven standard bearer.

A word of caution is in order: the Ada code in the examples has not been checked by a compiler, since there was none available at press time. There is no claim to correctness. The examples merely represent my best effort at interpreting the Ada reference manual [ADA 80].

Design Goals

In the introduction of the Ada manual [Ada 80], design goals are grouped in three categories:

Reliability

Programming as a human activity

Efficiency.

The following subgoals are spawned by program reliability:

Readability

English-like constructs

Avoidance of error-prone notation

Separate compilation of program units.

The COBOL 1974 standard [ANSI74] does not include design goals, and I therefore assume that the 1968 goals are still valid. The first two Ada reliability subgoals coincide with well known advantages of COBOL [KREK79]. Readable, English-like notation has remained a principal design goal since the 1968 COBOL standard. COBOL added separate compilation in 1974.

Other goals in the 1968 COBOL standard are:

Expandability

Problem orientation (to data processing)

Machine independence.

The following chart facilitates a comparison:

Goal	Ada	COBOL
Reliability	X	
Readability	X	X
English-like	X	X
Error resistant notation	X	
Separate compilation	X	(provided)
Programming as a human activity	X	
Efficiency	X	X
Expandability		X
Problem Orientation		X
Machine Independence	X (Implicit)	X

Hoare disputes the goal of facilitating future expansion [HOAR73]. Yet COBOL explicitly provides for expansion by its eleven module structure and provision for implementation in 2 or 3 stages. Thus COBOL is actually a family of languages under a common name [KREK79]. This situation is at odds with the goal of portability.

Ada, in contrast, discourages modifications. Revised Ironman [IRON74] decrees, "There shall be no subset or superset implementations." To a large extent the success of this concept will depend on the degree of enforcement by DoD. The road may be bumpy, because many other HOL's have over a period of time developed into a class of languages sharing the same name. Krekel cannot resist noting the analogy of a single disease which encompasses a collection of symptoms [KREK79].

The Ada Reference Manual [ADA 80] cites several subgoals based on consideration of programming as a human activity:

- to develop as few concepts as necessary.
- to avoid excessive involutions of the few concepts.
- to develop language constructs which correspond to the intuitive expectations of the programmer.

The first two subgoals resemble the ALGOL63 major objective of "orthogonality", but the last may be unique to Ada. There is no indication, though, of just how the programmer's intuitive expectations will be met [KREK79].

The goals of safety and ease of training are not claimed by Ada but are worthy of mention. Safety refers to the ability to discover syntax and other errors before they result in more serious problems.

Ada strong typing allows the compiler to determine data type compatibility, precluding serious and confusing run time errors. The idea of "type" is an imposition of structure on data. A type describes the set of values that objects of the type may take on, and the set of operations that can be performed on them. "Strong typing" is a characteristic of languages which rigidly enforce type rules. An example of enforcement is precluding the assignment of a floating point value to an integer variable, unless the floating point value is first explicitly converted to a value of integer type.

Pascal, of which Ada is a derivative, embraced training as a principal design goal. The idea was to produce a language "suitable to teaching programming as a systematic discipline" [WIRT78]. Pascal appears to have accomplished this goal, since it is now commonplace in the undergraduate curriculum. The jury for Ada is still in session, but some of the pioneers have had some difficulty teaching Ada. The most common complaints are:

- it's tough teaching a programming language which has no compiler

- Ada has so many features that it takes a long time to cover them

- Many Ada concepts are foreign to COBOL/FORTRAN programmers, e.g. packages, tasks, generics, information hiding, user defined types and strong typing.

On the brighter side, most educators I have contacted report that students are enthusiastic about Ada when they become aware of its advantages as a design tool. Accordingly, the most successful

instructors have used a "give the big picture first" approach. For example, LeBlanc at Georgia Institute of Technology introduces Ada via a series of topics related to overall system design methodology: program structure (packages), information hiding, abstract data types, separate compilation, management of program development.

Both COBOL and Ada claim efficiency as a design goal, but without elaboration on how to achieve it. The Ada concept was to examine every proposed construct with regard to present implementation techniques. Any construct with an unclear implementation or which required "excessive" machine resources was rejected [KREK79].

Krekel criticizes Ada design goals on these issues [KREK79]:

1. The problem domain is unclear. Reading samples from the Rationale for Ada [ICH79B] is the only way to understand the intended application.

2. Tradeoffs between design goals are not noted in Ada reports, e.g.

- conciseness of language definition vs. completeness and understandability

- redundancy vs. providing only one construct for each concept

- user convenience vs. efficiency.

3. No measures are provided to judge accomplishment of the design goal.

DeMarco [DEMA80], using the term, "language purity" as a design issue, criticizes Ada for extensions to Pascal which, though reasonable, are not essential. For example:

- Ada built in type STRING (in Pascal you can define your own such type)

- ASSERT verb (in Pascal, this feature could be user defined)

- ELSEIF (No big deal; the ELSE IF in Pascal does as well)

- The RAISE statement (a kind of Pascal alias for GO TO)

The conceptual language used by a programmer consists of the formally defined programming language plus a collection of user defined tools (extensions) expressed in that formal language. Ada anticipates certain programmer needs and provides a number of tools which are left to the programmer by Pascal. According to DeMarco, anticipating programmer needs is almost impossible, so the spareness of Pascal is a better approach. A lean language eases implementation, training, and programming [DEMA80].

Dijkstra in [DIJK78] issues a scathing, rhetorical indictment of Ada design goals, claiming that the Department of Defense didn't know "what it was asking for, and why." He sees Ada as a misguided attempt to "improve a compromise"(Pascal) by insisting that Ada better accomplish conflicting design goals. He also criticizes the apparent blurring of the distinction between the language and its implementation. He fears that Pascal may turn out to be an improvement over its successor -- Ada.

Overview of COBOL

At the highest level, a COBOL program consists of four divisions: IDENTIFICATION, ENVIRONMENT, DATA and PROCEDURE. The IDENTIFICATION Division identifies the program and its author. The ENVIRONMENT

Division prescribes machine dependent characteristics. It must be revised if the program is transferred to a new machine.

The DATA Division allows (and requires) the programmer to define the format and logical structure of his files, records and variables. These declarations are machine independent unless special machine dependent data representations are needed for efficiency. For example, COMPUTATIONAL representation for numeric items can take several different forms, depending on the internal machine representation desired by the programmer. Most data are defined as numeric or character (text) format:

```
77 EMPLOYEE_NUMBER PICTURE 9999. Numeric format
77 EMPLOYEE_NAME PICTURE XXXX.  Character format
```

Such data objects are considered by the programmer and the COBOL language to be in numeric/character format wherever they appear. This form of data object declaration helps establish easy to read formats for record input/output. Thus the DATA Division contains declaration of data objects and their types, but it does not allow type declarations apart from data objects.

The PROCEDURE Division contains the executable statements which are to be performed at execution time. This section resembles "the program" in a language like FORTRAN or BASIC.

COBOL was designed to automate manual procedures characterized by a series of small, well defined operations on data items. This is probably the rationale for the emphasis on COBOL readability at the sentence level -- each step in a COBOL program can be described in a near-English sentence, e. g.

ADD OVER_TIME TO NORMAL_EARNINGS GIVING TOTAL_EARNINGS.

This does achieve a measure of readability, but many modern day language experts would call it verbose. Today the emphasis in language design is on simplicity and generality ("orthogonality"). COBOL English-like syntax has been accomplished largely through liberal selection of reserved words, about 300 in all [ANSI74]. The "programming by selection from a menu" philosophy is resoundingly underscored by the approved addition of 93 new reserved words to the next revision of COBOL [COBO80]. Each is used in certain contexts to produce a quite clear result. The problem with this is that elegance and orthogonality are sacrificed -- that is, there are numerous special cases or exceptions to the rule. What works in one context does not in another. There are also more rules to learn. Modern theory extolls the importance of having the fewest possible features so as to make the language easier to master.

Features Ada Implements Directly: Arithmetic

The comparison of Ada and COBOL is organized in four categories: COBOL features Ada provides directly, features Ada emulates, Ada advantages, and things Ada does not do well. In the first category is arithmetic.

Ada syntax does not provide a structure to compete with certain COBOL English-like arithmetic statements, such as:

ADD OVER_TIME TO REGULAR_EARNINGS GIVING TOTAL_EARNINGS.

The Ada equivalent is simply the assignment statement:

```
TOTAL_EARNINGS := OVERTIME + REGULAR_EARNINGS
```

I nave found no significant evidence to support or detract from the importance of the COBOL approach. Since many programming languages seem to satisfy users with the assignment statement, I conclude that the Ada approach is satisfactory.

COBOL arithmetic may be accomplished most succinctly by the COMPUTE statement, e.g.

```
COMPUTE OVERTIME_EARNINGS = ((HOURS - 40.0) * 1.5) * PAY_RATE.
```

The COMPUTE statement allows assignment of the value of an arbitrarily complex expression to the variable on the left of the equals. Ada provides the same capability with the assignment statement, e.g.

```
OVERTIME_EARNINGS := ((HOURS - 40.0) * 1.5) * PAY_RATE;
```

I take note here of the clearer distinction in Ada notation between assignment (:=) and boolean operator (=). COBOL uses "=" for both purposes.

COBOL provides a ROUNDING option (an alternative to truncation) and an ON SIZE ERROR option which provides a limited programmer defined exception handling capability. For example:

```
COMPUTE AMOUNT ROUNDED = QUANTITY * PRICE ON SIZE ERROR PERFORM
OF_ERROR.
```

Ada can emulate this feature with user defined functions which allow the following equivalent statement:

```
COMPUTE(AMOUNT,QUANTITY*PRICE,ROUNDED);
```

This is rather awkward and is not recommended. A more convenient Ada solution is:

```
COMPUTE_ROUNDED(AMOUNT,QUANTITY * PRICE);
```

Better yet, perhaps is a two step approach:

```
AMOUNT := QUANTITY * PRICE;
ROUND(AMOUNT);
```

Or, one final variant:

```
AMOUNT := ROUND(QUANTITY *PRICE);
```

This last option is most in the spirit of Ada.

The implementation of Ada fixed point arithmetic through assignment statements or programmer defined functions, as in the above examples, requires a solution to a problem of types. Multiplication and division of fixed point values produce a result of greater, but undefined, accuracy. Thus the result has a different type than the operands. Explicit conversion to a user-defined fixed point type is necessary. Two solutions are obvious. One approach is to use built in type conversion functions associated with user defined types:

```
AMOUNT := AMOUNT_TYPE(QUANTITY * PRICE);
```

Another idea (which is impractical in large programs) is to overload the '*' and '/' operators by programmer defined implementations which would produce a result having the same type as the operands:

```
function '*' (X,Y :  AMOUNT_TYPE) return AMOUNT_TYPE;
```

This provides the illusion of implicit type conversion, but functions for '*' and '/' would have to be defined for every fixed point type in the program. Alternatively, the problem could be solved by a generic program unit (see the section on abstraction for an explanation of the Ada package):

```
generic
  type FIXED is delta <>;
package FIXED_POINT_OPN is
  function '*' (U,V : FIXED) return FIXED;
  function '/' (U,V : FIXED) return FIXED;
end FIXED_POINT_OPN;
```

Given one declaration of '*' and '/' functions for generic fixed point types, the programmer must instantiate the functions for the particular fixed point types used in the program:

```
PAY_TYPE_OPN is new FIXED_POINT_OPN(PAY_TYPE);
TOTAL_COST_TYPE_OPN is new FIXED_POINT_OPN(TOTAL_COST_TYPE);
```

Even now there are user defined operations only for operands of the same type. Mixing different types in a multiplication or division operation still requires an explicit type conversion, e.g.:

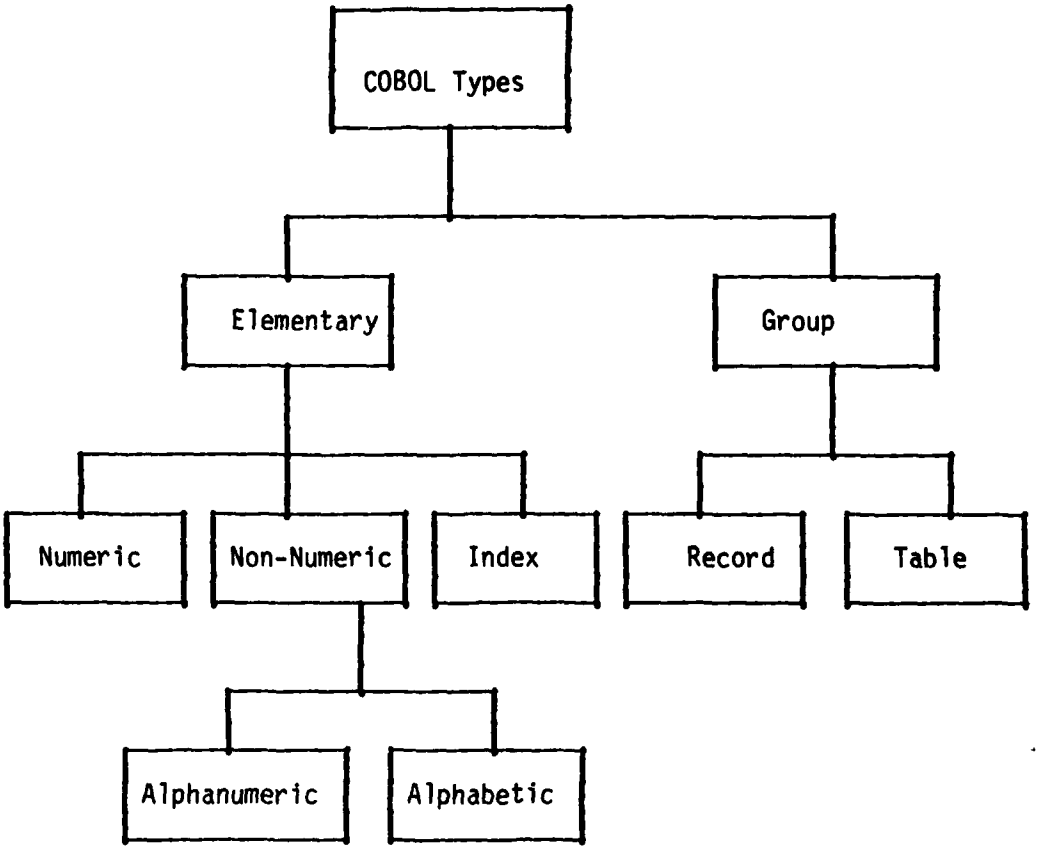
```
P :  PAY_TYPE;
T :  TAX_RATE_TYPE;
...
TAX := PAY_TYPE(T)*P;
```

The formerly presented method of explicit conversion would be preferred by most programmers, since it contributes to readability and simplicity.

The COBOL exception handling option ON SIZE ERROR is associated with the statement generating the exception. Ada exception handling facilities are more extensive and are discussed elsewhere. Since exception handlers in Ada are associated with a program unit, there can be one set of arithmetic exception handlers at the highest level. Alternatively the programmer can include a handler in any program unit which contains arithmetic statements.

Records

One of the most important, extensively used COBOL features is the record and its associated operations, particularly the MOVE, READ and WRITE. Records associated with files (and hence the READ and WRITE operations) will be discussed later as part of the I/O features, but for the moment the focus is on "working storage" records. Scalar variables and records may be declared in the working storage section, in analogous fashion to the declaration of variables and records in Ada. COBOL has a fixed menu of data types which may not be extended by the user. These are not types in the modern sense; they merely allow a hierarchial structuring (of elementary items) which corresponds to a character string. User defined types are not allowed. Available COBOL types may be described by the following diagram:



There is another important distinction from modern types: the compiler does not prevent type violations. Run time checking is necessary, and many checks are generated by the compiler. Yet COBOL lacks features which would assist the programmer in doing the job. Some type violations may escape both compile-time and run-time checks. For example, the declaration

```
77 EMPLOYEE_NUMBER PICTURE 9999.
```

does not prevent transferring 'DUMB', a non-numeric character string, to EMPLOYEE_NUMBER at run time. The programmer could insert his own type checking statement to determine whether the transfer is legal,

e.g.

```
IF EMPLOYEE_NUMBER IS NOT NUMERIC PERFORM TYPE_ERR_ROUTINE.
```

This paper is not intended to indict COBOL, so let us observe the brighter side, that for the supported data types COBOL provides a convenient syntax for data definition.

In COBOL, records may be defined in hierarchial fashion, with "group names" referring to a collection of subordinate elements. The Ada record is quite similar. Examples:

COBOL

```
01 TIME_CARD.
   03 EMPLOYEE_DATA.
      05 EMPLOYEE_NO PICTURE 9(4).
      05 EMPLOYEE_NAME PICTURE X(20).
   03 NORMAL_EARNINGS PICTURE 999V99.
   03 OVER_TIME PICTURE 999V99.
```

Ada (using type definition and anonymous types)

```
type TIME_CARD_TYPE is
  record
  EMPLOYEE_DATA :  record
                    EMPLOYEE_NO :  INTEGER range 0 .. 9999;
                    EMPLOYEE_NAME :  STRING(1 .. 20);
                   end record;
  NORMAL_EARNINGS :  delta .01 range 0.0 .. 999.99;
  OVER_TIME :  delta .01 range 0.0 999.99;
  end record;
TIME_CARD : TIME_CARD_TYPE;
```

The above definition is remarkably similar to that of COBOL, but the following declaration is more in the spirit of Ada, and it avoids redundant anonymous type declarations.

Ada (using type definition - all types named)

```
type EMPLOYEE_NO_TYPE is INTEGER range 0 .. 9999;
type EMPLOYEE_NAME_TYPE is STRING(1 .. 20);
type EMPLOYEE_DATA_TYPE is
  record
    EMPLOYEE_NO : EMPLOYEE_NO_TYPE;
    EMPLOYEE_NAME : EMPLOYEE_NAME_TYPE;
  end record;
type NORMAL_EARNINGS_TYPE is delta .01 range 0.0 .. 999.99;
type TIME_CARD_TYPE is
  record
    EMPLOYEE_DATA : EMPLOYEE_DATA_TYPE;
    NORMAL_EARNINGS : NORMAL_EARNINGS_TYPE;
    OVER_TIME : NORMAL_EARNINGS_TYPE;
  end record;
     ...
TIME_CARD : TIME_CARD_TYPE;
```

Use of named constants further enhances the Ada declaration by increasing readability and expediting later changes to type definitions. Example:

```
EMPLOYEE_NO_MAXIMUM : constant INTEGER := 9999;
EMPLOYEE_NAME_MAXLENGTH : constant INTEGER := 20;
NORMAL_EARNINGS_MAXIMUM : constant := 999.99;
type EMPLOYEE_NO_TYPE is INTEGER range 0 .. EMPLOYEE_NO_MAXIMUM;
type EMPLOYEE_NAME_TYPE is STRING (1 .. EMPLOYEE_NAME_MAXIMUM);
type NORMAL_EARNINGS_TYPE is delta .01 range 0.0 .. NORMAL_EARNINGS_MAXIMUM;
```

The remainder of the declaration is the same as the previous example. Notice the progressive approach available in Ada for type declarations. More complicated types can be defined in terms of constants and other previously defined types. The programmer may easily change a type definition even if it has impact throughout the program. For example, redeclaring EMPLOYEE_NAME_MAXLENGTH achieves the desired change, even though numerous types and variables depend on this item.

The Ada package may also be used to declare a data object.

Ada (using package)

```
package TIME_CARD is
  EMPLOYEE_DATA : record
                    EMPLOYEE_NO : INTEGER range 0 .. 9999;
                    EMPLOYEE_NAME : STRING (1 .. 20);
                  end record;
  NORMAL_EARNINGS : delta .01 range 0.0 .. 999.99;
  OVER_TIME : delta .01 range 0.0 .. 999.99;
end TIME_CARD:
```

Reference to COBOL and Ada record/package elements is similar as the examples below point out. The Ada references are correct for all the above example declarations.

COBOL Reference to Record Element

EMPLOYEE_NO of EMPLOYEE_DATA of TIME_CARD

(Note: If element names are unique the above may be abbreviated to: EMPLOYEE_NO)

Ada Reference to Record/Package Element

TIME_CARD.EMPLOYEE_DATA.EMPLOYEE_NO

Note: The clause "use TIME_CARD" allows a reduction in qualification:

EMPLOYEE_DATA.EMPLOYEE_NO.

Further reduction is possible with "use TIME_CARD.EMPLOYEE_DATA" which allows the same abbreviation as the COBOL example:

EMPLOYEE_NO .

One disadvantage of the package as a data structure is that there is no built in way to access or manipulate the data structure as a whole, and there is no way to define more than one object of the same type.

Unlike the Ada package example and the COBOL record, the Ada type definition does not establish a data object. Therefore it is not meaningful (and is illegal in Ada) to use a type name in the role of a variable. Instead, a type declaration creates the format of a data object, allowing subsequent creation of objects of that particular type (format).

Illegal use of type name

```
type EMPLOYEE_NO_TYPE is INTEGER range 0 .. 9999;
EMPLOYEE_NO_TYPE := 56;  -- illegal --
```

An advantage of the type name is the ability to declare any number of different data objects of each type. In a COBOL application requiring numerous record definitions, this ability could be used as a valuable shorthand, readability and error prevention feature. Record elements intended to have the same type can be so declared by referring to the appropriate type name. Consider the following example, where the EMPLOYEE_NORMAL_EARNINGS_TYPE is needed in several different record types:

```
EMPLOYEE_NORMAL_EARNINGS_TYPE is delta .01 range 0.0 .. 999.99;
type SPECIAL_EARNINGS_TYPE is
  record
    EMPLOYEE_BONUS : EMPLOYEE_NORMAL_EARNINGS_TYPE;
     ...
  end record;
type MONTHLY_STATISTICS_TYPE is
  record
    EMPLOYEE_AVG_SALARY : EMPLOYEE_NORMAL_EARNINGS_TYPE;
     ...
  end record;
```

This Ada feature greatly simplifies changes of record format,

particularly when one needs to alter a single record element type which appears in numerous different records (this is a common situation in COBOL programs).

COBOL allows initialization of working storage records, a feature often used to establish titles and labels:

COBOL Record With Initial Values

```
01 TIME_CARD_LIST_TITLE.
   03 FILLER PICTURE X(20) VALUE SPACES.
   03 FILLER PICTURE X(15) VALUE 'EMPLOYEE NUMBER'.
   03 FILLER PICTURE X(5) VALUE SPACES.
   03 FILLER PICTURE X(13) VALUE 'EMPLOYEE NAME'.
   03 FILLER PICTURE X(7) VALUE SPACES.
   03 FILLER PICTURE X(15) VALUE 'NORMAL EARNINGS'.
   03 FILLER PICTURE X(5) VALUE SPACES.
   03 FILLER PICTURE X(17) VALUE 'OVERTIME EARNINGS'.
```

Ada has a corresponding means of initializing package data objects and establishing default initialization of data types:

Ada Data Type With Initial Values

```
Function SPACES(HOW_MANY : INTEGER) return STRING;
type TIME_CARD_LIST_TITLE_TYPE is
  record
    FILLER1 : STRING(1 .. 20) := SPACES(20);
    FILLER2 : STRING(1 .. 15) := 'EMPLOYEE NUMBER';
    FILLER3 : STRING(1 .. 5) := SPACES(5);
    FILLER4 : STRING(1 .. 13) := 'EMPLOYEE NAME';
    FILLER5 : STRING(1 .. 7) := SPACES(7);
    FILLER6 : STRING(1 .. 15) := 'NORMAL EARNINGS';
    FILLER7 : STRING(1 .. 5) := SPACES(5);
    FILLER8 : STRING(1 .. 17) := 'OVERTIME EARNINGS';
  end record;
  ...
TIME_CARD_LIST_TITLE : TIME_CARD_LIST_TITLE_TYPE;
```

Notice the use in the above example of a programmer defined function to handle the built in COBOL literal SPACES. My convention in this paper is not to define the inner workings of functions and procedures so as not to distract the reader with details. COBOL has several built in literals to simplify the programmer's coding job and contribute to readability. The COBOL VALUE clause also allows initialization of variables in the data declaration to any constant value. The Ada function is a much more powerful technique since it allows similar benefits for any string of characters desired -- not just spaces. The only difference from COBOL literals appears to be the need to specify the length of the string of characters required.

This is not significant, for the necessary number has already been specified earlier in the statement. Use of the SPACES function elsewhere in the Ada program, as in an assignment statement, requires the programmer to be aware of the size of the string to which characters are being transferred.

Ada Package With Initial Values

```
package TIME_CARD_LIST_TITLE is
FILLER1 : STRING (1 .. 20) := SPACES(20);
    ...
FILLER8 : STRING(1 .. 17) := 'OVERTIME EARNINGS';
end TIME_CARD_LIST_TITLE;
```

The FILLER name is used in COBOL to designate an anonymous variable, one which the programmer has no need to reference. It is commonly used in the definition of record elements. Ada has no corresponding reserved word, but its visibility rules provide a similar convenience. COBOL needs the FILLER feature to avoid unintentional naming conflicts, since with few exceptions all variables are global. In Ada a record element name will not accidently "hide" or conflict with another as it would in COBOL. Example:

COBOL Naming Conflict

```
01 TIME_CARD_DETAIL_LINE.
   03 EMPLOYEE_NAME_PICTURE X(20).
   03 EMPLOYEE_TOTAL_EARNINGS PICTURE 999.99.
     ...
77 EMPLOYEE_TOTAL_EARNINGS PICTURE 99.99.
     ...
MOVE 25.00 TO EMPLOYEE_TOTAL_EARNINGS.
```

(Note: At this point in the program we have conflicting names visible).

Ada programs consist of one or more modules. Identifiers declared in a module are local to that module. The programmer is free to ignore identifiers declared externally unless he needs to use them. When he wishes to use "imported" names he can consciously avoid hiding them with local redeclarations. This is a simplistic explanation of Ada visibility (scope) rules (see [ADA 80]).

Features Ada Emulates: The COBOL MOVE

A 1980 AIRMICS Software Science project [GABR80] includes the analysis of production COBOL programs using an automated COBOL analyzer developed at Purdue University. One of the (not surprising) findings is that the MOVE statement is one of the most used. The MOVE in COBOL does much of the job of the assignment statement in other languages. MOVE is perhaps most frequently used to transfer contents of records (either in their entirety or by components) to other records. COBOL associates with MOVE an elaborate automatic type conversion facility as indicated in the following table [ANSI74]:

Category of Receiving Data Item

Category of Sending Data Item	Alphabetic	Alphabetic Edited or Alphanumeric	Numeric Non-Integer Numeric Edited
Alphabetic	yes	yes	no
Alphanumeric			
Alphanumeric Edited	yes	yes	no
Numeric Integer	no	yes	yes
Numeric Non-Integer	no	no	yes
Numeric Edited	no	yes	no

Ada is perhaps at the other extreme with regard to automatic type conversion. This is no accident, because the technical requirements for Ada specified [IRON79]: "There shall be no implicit conversions between types." Differences in range, precision and scale are not construed as differences in type, yet there is no implicit truncation or rounding in integer and fixed point computations. Thus explicit conversion is necessary to constrain the result of a fixed point multiply or divide to the same type as the operands.

A number of language design experts support the Ada approach, but seasoned FORTRAN and COBOL programmers may cry in anguish over the tedium of worrying about what they perceive are unimportant details about data types. Though it seems rather autocratic, Wirth [WIRT74] advocates giving programmers what they need to solve their problems,

not what they say they want, because many programmers would insist on (perish the thought) typeless operands. Hoare [HOAR73] reinforces Wirth's emphasis on the value of strong typing by advocating, "the complete avoidance of any form of automatic type transfer, coercion, or default convention."

Gannon and Horning [GANN75] preach the virtue of redundancy as in type declarations. They demonstrate the great contribution to readability, error detection and program reliability. The context of each use of a data item can be checked by a compiler for agreement with its declared type. Ability to declare restrictions of variables to subranges of a parent type (as in Ada) allows the compiler to optimize storage allocation and, more important, permits easy detection of violations at run time which could otherwise be difficult logic bugs.

Suppose a "COBOL to Ada convert" does not care what the language design experts think. Suppose he insists on the next best thing to automatic type conversion that Ada can offer. He should then consider (extensively) overloading a programmer defined MOVE procedure. MOVE could take the general form:

```
MOVE(SOURCE_NAME,DESTINATION_NAME);
```

Generic packages can be used to advantage here, but a separate instantiation will be necessary for each combination of operand types (source and destination) which may occur in the program. A rather tedious chore, this method will nonetheless make our COBOL diehard happy. Let us consider an example implementation of a MOVE using fixed point source and destination. Assume the following declarations:

```
type NORMAL_EARNINGS_TYPE is delta .01 range 0.0 .. 999.99;
type HIGH_PRECISION_TYPE is delta .0001 range 0.0 .. 999.999;
NORMAL_EARNINGS : NORMAL_EARNINGS_TYPE;
YEAR_TO_DATE_AVERAGE : HIGH_PRECISION_TYPE;
```

Implementation of the move could take this form:

```
generic
  type SOURCE_TYPE is delta <>;
  type DESTINATION_TYPE is delta <>;
package FIXED_POINT_MOVE is
  procedure MOVE(S : in SOURCE_TYPE, D : out DESTINATION_TYPE);
end;
package body FIXED_POINT_MOVE is
  D := DESTINATION_TYPE(S);
end;
```

To instantiate, we use declarations like:

```
package MOVE1
  is new FIXED_POINT_MOVE(SOURCE_TYPE => NORMAL_EARNINGS_TYPE,
                          DESTINATION_TYPE => HIGH_PRECISION_TYPE);
package MOVE2
  is new FIXED_POINT_MOVE(SOURCE_TYPE => HIGH_PRECISION_TYPE,
                          DESTINATION_TYPE => NORMAL_EARNINGS_TYPE);
```

Now, finally, the programmer may use the MOVE procedure in either direction and achieve the desired type conversion:

```
MOVE(NORMAL_EARNINGS , YEAR_TO_DATE_AVERAGE);
MOVE(YEAR_TO_DATE_AVERAGE , NORMAL_EARNINGS);
```

Even though multiple definitions of MOVE will exist, the compiler will select the proper one by examining the types of the operands. Consider the impracticality of this approach. To provide for all possible moves to/from each type, the number of MOVE procedure instantiations will be proportional to the square of the number of data types. (100 data types => about 10,000 MOVE instantiations!) One could develop a utility program to scan an Ada program and automatically generate appropriate MOVE instantiations to handle all possibilities. A more reasonable alternative is programmer instantiation of MOVE procedures on an "as necessary" basis.

An even better idea is to remain aware of operand types used in the program and employ Ada built in type conversion functions. For example the previous moves could be accomplished using just the following statements:

```
YEAR_TO_DATE_AVERAGE := HIGH_PRECISION_TYPE(NORMAL_EARNINGS);
NORMAL_EARNINGS := NORMAL_EARNINGS_TYPE(YEAR_TO_DATE_AVERAGE);
```

The COBOL "alphanumeric MOVE" can be handled in Ada in similar fashion to the "fixed point MOVE" previously described. An instantiation of an overloaded MOVE procedure would be necessary for each combination of strings of various lengths. Built in type conversion functions could also be used.

COBOL also allows MOVE of a numeric item to an alphanumeric item. In Ada this corresponds to a move of an integer, fixed point or floating point type to a string type. Built in type conversion will not suffice. The best way to accomplish this MOVE seems to be the generic package with multiple instantiations. Example:

```
type NORMAL_EARNINGS_TYPE is delta .01 range 0.0 .. 999.99;
type NORMAL_EARNINGS_DISPLAY_TYPE is STRING(1 .. 20);
NORMAL_EARNINGS : NORMAL_EARNINGS_TYPE;
NORMAL_EARNINGS_DISPLAY : NORMAL_EARNINGS_DISPLAY_TYPE;
```

We want to implement MOVE such that

```
MOVE(NORMAL_EARNINGS,NORMAL_EARNINGS_DISPLAY)
```

will accomplish the fixed point to string conversion. The following generic package could do the job:

```
generic
  type FIX_TYPE is delta <>;
  type STRING_OBJECT_TYPE is STRING(NATURAL range <>);
procedure MOVE_IT(FIX_SOURCE: in FIX_TYPE;
                  STRING_DESTINATION: in out STRING_OBJECT_TYPE) is
begin
  STRING_DESTINATION := FIX_TYPE'IMAGE(FIX_SOURCE);
end MOVE_IT;
```

An appropriate instantiation in this case is procedure MOVE is

```
new MOVE_IT(FIX_TYPE => NORMAL_EARNINGS_TYPE,
    STRING_OBJECT_TYPE => NORMAL_EARNINGS_TYPE);
```

This is an appropriate time to observe what may already be obvious -- that Ada packages and other constructs provide an explicitly coded solution to built in COBOL features which are implicitly implemented by the COBOL compiler. This Ada scheme introduces an extra degree of source code complexity. The unfavorable impact can be minimized, though, by the provision of a generous set of predefined, pretested, precompiled Ada packages in the Ada library at an installation. Ada's facilities for modular programming make this a reasonable approach. The programmer need only be aware of the external interface of a package. Details of the implementation are of no concern.

Sort Capability

Next to the record, the sort capability must be the most useful feature for business data processing. Few programs indeed can do without it. In COBOL the programmer may define one or more sort files in the DATA Division. Example:

```
SD SORT_FILE.
   ...
01 SORT_REC_1.
   03 EMPLOYEE_DATA_S.
      05 EMPLOYEE_NO_S PICTURE 9(4).
      05 EMPLOYEE_NAMES PICTURE X(20).
   03 NORMAL_EARNINGS_S PICTURE 999V99.
   03 OVER_TIME_S PICTURE 999V99.
```

```
01 SORT_REC_2.
   05 ACCOUNT_NUMBER_S PICTURE 9(6).
   05 NAME_S PICTURE X(20).
   05 LOAN_AMOUNTS PICTURE 9(5)V99.
```

The above record layouts bear suspicious resemblance to those presented later in the discussion of records. In fact they are the same except for the "S" suffix on each identifier for readability. This allows us to easily sort the files consisting of TIME_CARD or LOAN_RECORD records on any of the elements of the records. Example:

```
SORT SORT_FILE ON ASCENDING KEY ACCOUNT_NUMBER_S NAME_S
  USING LOAN_MASTER_FILE
  GIVING NEW_MASTER_FILE.
```

This simple command does a big job, including opening the source and target files, reading the source file, performing the specified sort, and storing the result in the target file.

How shall Ada compete with the powerful SORT verb? The package comes to the rescue. Let us heed modern language design theory which recommends encapsulating data structures and the operations defined on those structures. We shall resurrect the previously defined package which included a definition of the LOAN_RECORD_TYPE and add to it a procedure to sort files containing records of that type.

```
package LOAN_MASTER_FILE_PACKAGE is
  type LOAN_RECORD_TYPE is
    record
    ACCOUNT_NUMBER : range 0 .. 999999;
    NAME : STRING(1 .. 20);
    LOAN_AMOUNT : delta .01 range 0.0 .. 99999.99;
    end record;
type UP_DOWN_TYPE is (ASCENDING, DESCENDING);
LOAN_RECORD : LOAN_RECORD_TYPE;
procedure WRITE (RECORD : in  LOAN_RECORD_TYPE);
procedure READ(RECORD : out  LOAN_RECORD_TYPE);
procedure SORT(INPUT_FILE : in FILE_NAME_TYPE;
               OUTPUT_FILE : out FILE_NAME_TYPE;
               UP_DOWN : in UP_DOWN_TYPE;
               ACCOUNT_NUMBER_SORT_PRI : SORT_ORDER := 1;
               NAME_SORT_PRI : SORT_ORDER := 2;
               LOAN_AMOUNT_SORT_PRI : SORT_ORDER := 3);
end LOAN_MASTER_FILE_PACKAGE;
```

Below is a sample use of the package:

```
SORT(INPUT_FILE => LOAN_TEMP_FILE,
     OUTPUT_FILE =>LOAN_SORT_FILE,
     UP_DOWN => ASCENDING,
     ACCOUNT_NUMBER_SORT_PRI => 2,
     NAME_SORT_PRI => 1);
```

Notice that in Ada not all procedure parameters have to be provided in

the call; default values will be used for those omitted. The above accomplishes a sort of the LOAN_TEMP_FILE using the ACCOUNT_NUMBER element as the key. This is not a particularly elegant solution -- there must be something much better -- but it does come close to the power of the COBOL SORT. Except for the package body (which includes details of the SORT procedure) the programmer does not have much more work to do than in COBOL to "set up" the SORT procedure for use. The SORT procedure call is almost as convenient to use as the COBOL version and is nearly as readable. The most awkward part of the solution presented here is the need to assign "priority numbers" to the record elements instead of merely listing the sort key and defining ascending or descending priority. But surely a clever Ada programmer can improve on this solution.

Database Considerations

All major existing standard Army MIS are based on sequential files, based on the magnetic tape storage medium, but USACSC is committed to incorporating DBMS technology in new systems and major rewrites. New systems will take better advantage of random access storage devices. An ongoing large scale redesign of the Standard Army Financial System will be the first USACSC attempt at incorporating a DBMS in a large MIS.

An attractive benefit of COBOL is the existence of a generous collection of COBOL oriented DBMS and database development tools. The Data Base Task Group of the Conference on Data Systems Languages has developed a data definition language (DDL) and a data manipulation language (DML) which are compatible with COBOL. Popular existing COBOL-oriented DBMS include IMS, TOTAL, SVS200, IDMS, ADABAS and

DMS-1100. There are a number of automated tools which help develop schemas and sub schemas. Many produce COBOL Data Division code.

In time there will be DBMS for Ada. One ongoing effort to develop an Ada oriented DBMS is that of Computer Corporation of America, sponsored by DoD and the Navy [POTT80]. Called ADAPLEX, it provides a set of database commands to be embedded in Ada. The database facilities will be designed as Ada packages. ADAPLEX should be available in the mid 1980's. Existing COBOL-oriented DBMS and tools can be adapted to Ada in a straightforward manner.

Ada Advantages: Strong Typing

According to Gannon [GANN76] the aim of reliable programming is to reduce the number of errors in delivered software. This goal may be achieved by preventing programmer errors or by increasing the percentage of errors which are detected and corrected before the final product is complete. Strong typing, an Ada design goal, can help in both departments.

A data type determines what operations are permitted on operands of that type. The two biggest advantages of data types are abstraction and authentication [MORR73]. A data type hides the implementation details for objects of that type, so a programmer can (and must) deal with an abstraction which facilitates solving the problem at hand. The programmer can deal with objects like "stack", "matrix", or "color" rather than collections of machine memory words or bit patterns within words. Authentication, often called type checking, insures that invalid operations cannot be performed [GANN76]. For example, authentication will prevent arithmetic on boolean

or string values. Authentication may be approached several ways in a programming language: static type checking, dynamic type checking, or no type checking.

Ada emphasizes static checking, which means that a data type is permanently associated with a variable when it is declared, and that operations are checked for validity at compile time. Dynamically typed languages allow the type of a variable to vary during program execution. Generally the variable type is defined as the type of the last value assigned to the variable. Typeless languages consider each operand as a bit pattern which is assumed to represent a value of the type that the operator requires. Assembler languages are usually typeless.

COBOL provides for declarations of variables but allows their types to vary, in many cases, through assignment or automatic type conversion. COBOL seems to be a cross between dynamically typed and typeless, but is probably best characterized as dynamic, since the few type checks that are made are deferred until run time.

Programmers who favor dynamically typed languages maintain that they are flexible to use and easy to implement, but they overlook a serious problem. Type violations (operators having operands of the wrong type) may be checked only at run time. Run time checks tend to slow execution. Worse, a run time check only detects errors in code actually executed, so a type violation in a piece of code not exercised during testing will survive to later plague the final product [GANN78].

Advocates of typeless languages often hail their allowing the programmer to take maximum advantage of the characteristics of the underlying machine [WIRT74]. Wirth says that the most prevalent programming trick is to pack several different kinds of data into a single machine word. He points out that this objective can more elegantly be accomplished in a statically typed language [such as Ada] using a record structure.

The system language for Project SUE [CLAR73] is a good example of the value of type checking in operating system design: type checking was the most valuable logic error prevention aid in the project.

Static typing is effective in error prevention. The declaration of data types is redundant, since the implied type of an operand may be determined from the context. This redundancy is advantageous, though, because it allows a compiler to check each use of an operand to insure that the operation is defined for the declared operand type.

Gannon conducted experiments and reached the conclusion that statically typed languages with explicit conversion provide early and reliable detection of errors [GANN76]. He observed 25 experienced programmers each solving several small but complex programs. The two programming languages used in the experiment provided static typing for some data types and dynamic typing for others. This was clearly documented in the language manuals. Occurrences of errors were logged for each run. Persistence of errors of a certain type was calculated as the number of occurrences divided by the number of errors of that type. During 815 program runs, 3937 occurrences of 1248 errors were recorded. 405 occurrences of 116 errors were caused by operands with

incorrect data types. 329 occurrences (of 104 errors) could have been precluded by compile time (static) checking.

Even in typed languages there remains the alternative of explicit or implicit type conversion, COBOL embellishes the latter. and this does make programming faster, since little attention need be given to type compatibility. But Hoare cites several disadvantages of implicit conversion:

- Errors are often masked by "nearly" correct results
- Run time overhead is a significant penalty
- Conversion rules complicate the language definition.

Certainly COBOL is guilty of the third disadvantage. For proof, take a look at the large table of conversion rules presented earlier.

Data Abstraction

Classical linguistic theory has established that the kind of language we use affects the way we think about solving problems [DILL77]. For example, a FORTRAN programmer probably seldom thinks of using a recursive procedure or a linked list, but these are everyday tools of the LISP programmer [WULF80].

Whether during development of a large programming project or during the maintenance phase, the most important factor affecting the difficulty of getting the job done is readability of the code. Ease of writing is not so important [WULF80]. To understand a piece of a program one must first understand what the program is to do, and most programming languages tend to emphasize how a problem was solved rather than what the program was supposed to do. Thus it is easier to

understand a queuing operation when it is defined as a separate, abstract operation than to determine that a sequence of statements modifying arrays happens to implement a queuing operation [WULF80].

A programming language serves three purposes [WULF80]: a design tool, a vehicle for human communication, and a means of instructing a computer. The first two purposes are the most significant, since they relate most strongly to possible reduction of costs in the maintenance phase. Over half the total cost of software development is often attributable to maintenance. As we shall see, data abstraction facilities are very helpful in improving design procedures and human communication.

The structured programming approach was adopted by USACSC in the 1970's in an effort to increase reliability of systems and reduce development and (particularly) maintenance costs [MITC80]. The structured programming concept includes a disciplined approach to system design, a management scheme and a methodology for programming [CARR78]. "Top down development" is a core philosophy. Systems are to be conceived of and described initially as a set of high level control modules and functional modules. As the design phase continues, one procedes to the detailed design of lower level functions. There appears to be no conceptual difference between this and Wirth's "stepwise refinement" concept [WIRT74]. The goal is to emphasize overall design by postponing concern over details of implementation. Structured programming, then, should take advantage of programming language features related to Wulf's first two purposes: a design tool and vehicle for human communication.

USACSC has a heavy investment in the structured programming approach [CARR78]:

- defining the theory and concept (1974-75)
- determining how to apply the technology in USACSC (1975-76)
- actual conversion to the approach (1976-77).

The Division Logistics System was the first major application of structured programming, and the use of this technology was deemed a success by most managers and programmers [CARR78].

In the Procedure Division COBOL provides facilities for structure of text and of control. Text may be structured by concatenation of sentences into paragraphs or paragraphs into sections. Control structures include [JACK76]:

- PERFORM can execute a paragraph or section which is not "in line"
- PERFORM can provide iteration (While/Do/For)
- GO TO provides transfer to the start of a paragraph or section
- GO TO DEPENDING ON gives a multiway branch.

The above features are collectively inadequate. Designers of COBOL, like those of most older programming languages, did not consider what design methodology would be used to write programs. As a result COBOL provides few aids and some hindrances to designing large systems [JACK76]. For example, all variable names are global; there is no way to restrict the use of a variable to a particular procedure [JACK76]. No variable is safe from access or alteration by instructions in any part of a program. In a large program this can create the same sort of mystery that often confronts users of FORTRAN COMMON. The situation is particularly troublesome to those attempting to use library procedures.

Certainly programming language data abstraction facilities (such as Ada provides) make the top down approach easier to implement. Indeed they provide significant advantages in the most important aspects of program development: decomposition, documentation and modification [LEBL80]. LeBlanc uses the term "data abstraction" in a broad sense which includes procedural abstraction as well. The most important step in program development is dividing the problem into smaller problems which are easy to understand and program. There are many ways to slice a pie, but an effective decomposition of a program requires:

- a natural, "meaningful" partitioning
- well defined interfaces between units
- hiding the details of the units from one another.

The natural partitioning depends on the skill of an experienced analyst, but languages which require formal definition of abstractions help fulfill the other two requirements.

The Ada package is the primary building block of Ada programs. It encapsulates data with the subprograms which operate on that data, thus specifying a collection of logically related computational resources. A package is normally stated in two parts: a package specification and a package body. The specification is the description of what the package is supposed to do. It may be considered a window through which the package is to be viewed. The Ada specification has all the information needed to use the facilities of the abstraction, and since it is the only way to access the abstraction

the details are hidden. Consider the following example from [BARN80]:

```
package STACK is            --specification
  procedure PUSH(X:REAL);
  function POP return REAL;
end;

package body STACK is       --body
  MAX:constant := 100;
  S:array(1..MAX of REAL;
  PTR:INTEGER range 0..MAX;
  procedure PUSH(X:REAL) is
  begin
      PTR := PTR + 1;
    S(PTR) := X;
  end PUSH;
  function POP return REAL is
  begin
    PTR := PTR - 1;
    return S(PTR + 1);
  end POP;
```

Though stacks are not often used in MIS programs, the example illustrates many key features of the Ada package. Notice the separation of the specification and body. If this package body is precompiled and only the source code for the specification is available, then users of the package may employ the procedures PUSH and POP but they need not know about, and cannot access, details of the stack implementation.

In many cases it is useful to provide limited access to a data structure. The Ada private type declaration establishes a type such that objects of this type have only the following operations defined on them outside the package in which the objects are declared: assignment, "=", and "/=". The most restrictive access is provided by a declaration of an object of a limited private type. In this case, only the operations explicitly provided by the programmer may be applied to the object outside the package in which it is declared. The following adaptation of an example from the Ada Reference Manual is a good illustration:

```
package I_O PACKAGE is
  type FILE_NAME is limited private;
  procedure OPEN(F: in out FILE_NAME);
  procedure CLOSE(F:in out FILE_NAME);
  procedure READ(F: in FILE_NAME; ITEM: out INTEGER);
  procedure WRITE(F: in FILE_NAME; ITEM: in INTEGER);
private
  type FILE_NAME is INTEGER := 0;
end I_O_PACKAGE;

package body I_O_PACKAGE is
  LIMIT : constant := 200;
  type FILE_DESCRIPTOR is record ...end record;
  DIRECTORY : array(1..LIMIT) of FILE_DESCRIPTOR;
  ...
  procedure OPEN(F : in out FILE_NAME) is ... end;
```

```
   procedure CLOSE(F : in out FILE_NAME) is ... end;
   procedure READ(F : in FILE_NAME;ITEM: out INTEGER) is ... end;
   procedure WRITE(F : in FILE_NAME;ITEM: in INTEGER) is ... end;
begin
   ...
end I_O_PACKAGE;
```

Given the above package, the programmer can declare objects of type FILE_NAME and can get a value for an object of this type by using the operation OPEN. The operations CLOSE, READ and WRITE are also available.

```
   A_FILE : FILE_NAME;      --establishes data object
   X : INTEGER;
   ...
   OPEN(A_FILE);            --gives A_FILE a value and opens the
                            --file having this value
```

The programmer cannot access, or take advantage of knowing, the details of the implementation of FILE_NAME. Therefore the package body is safe from tampering or abuse. Suppose a programmer found out that FILE_NAME was implemented as an integer. He or she then might attempt the following statements, all of which are illegal and would be flagged by the Ada compiler:

```
   A_FILE : FILE_NAME := 0; --assignment of an initial value
                            --is not an operation defined
                            --for type FILE_NAME
   A_FILE := 3;             --assignment not available
   A_FILE := A_FILE + 1;    --assignment, addition not available
   IF B_FILE > A_FILE then ... --boolean operations not available
```

The above examples show how package implementation details are

hidden from the outside world and are safe from intentional or accidental tampering, particularly clever programming tricks, which lead to insidious errors and portability problems. Perhaps "hiding" is a poor choice to describe this phenomena, since some software project managers believe some of their biggest problems spring from information hidden from them by their subordinates. To them, "information protection" would have a better connotation.

The package body implements the package specification. The two parts may be separately compiled and text of the package specification and body need not be contiguous.

Documentation should be a continuous process during development and not a separate phase or afterthought. Leblanc notes the important contribution of language abstraction facilities to the "development of well structured (self documenting) programs using stepwise refinement" [LEBL80]. This is the top down approach, which calls for initially defining high level abstractions for major functions. The process of implementing these abstractions results in the conception of more lower level abstractions, and the process continues until it is reasonable to implement the abstractions using built in language features or to take advantage of existing library abstractions [LEBL80].

Using Ada, a programmer can think and write initially in terms of package specifications in order to sketch a solution to a problem. The details of implementation (that is, the package bodies) may be deferred.

Names can be chosen freely in each package. For example there need be no programmer concern for using INSERT as a procedure name,

even if it has been used before as a procedure name. Even if two procedures with the same name are active (visible) at some point in a program there is no conflict as long as the procedure parameter types are distinct, as in the following case [ICHB80]:

```
procedure INSERT(E:ELEMENT);
procedure INSERT(E:ITEM);
```

In the rare event that parameter types are identical, then qualification or the RENAMES option can resolve naming conflicts:

```
procedure INSERT_QUEUE(E:ITEM) renames QUEUE.INSERT;
procedure INSERT_STACK(E:ITEM) renames STACK.INSERT;
```

There is no aliasing problem here as there is with the COBOL RENAMES, for the original (conflicting) names are considered invalid and illegal. In COBOL both the original and the new name can refer to the same data.

All software maintenance actions (fixes, enhancements) require a program to be changed. The ease of modification is directly related to the ability to localize the effects of the change [LEBL80]. The encapsulization of an abstraction such as the Ada package greatly assists in this localization. The formal definition of the abstraction (the Ada package specification) logically separates the abstraction implementation from other program units which use it. If the implementation is wrong, and must be changed to comply with its specification, the programmer may amend it while basking in the confidence that he is causing no unexpected side effects outside the abstraction. If the specification must be changed, then so must its

implementation, and all units which use this abstraction may need alteration. But there will be fewer changes required than there would if other units accessed the inner workings of the abstraction [LEBL80].

Most programmers are well aware of the advantages of procedures with parameters. They allow the same section of code to be used to process more than one set of data. Parameterizing abstractions extends the application in like manner. One well understood, debugged, general purpose abstraction can be established as several different instances by varying its parameters [LEBL80]. This requires less code and less mental effort than creation of numerous special case abstractions. Ada generic packages provide translation time parameterization using a general package definition. Consider the following generic package, which is a generalized version of the stack package presented earlier [BARN80]:

```
generic
  MAX:INTEGER;
  type ELEM is private;
package STACK is
  procedure PUSH(X:ELEM);
  function POP return ELEM;
end;
package body STACK is
  S:array(1..MAX) of ELEM;
  ...
end STACK;
```

A particular instance of a stack is declared by a "generic instantiation" which provides actual parameters. Example:

```
declare
  package REAL_STACK is new STACK(100,REAL);
  use REAL_STACK;  --allows shorthand reference
                   --to PUSH instead of REAL_STACK.PUSH
begin
  PUSH(X);
  ...
  X := POP();
end;
```

Both the size of the stack and the type of elements contained in the stack are parameters, so this piece of code may be instantiated to satisfy the need for stacks of various sizes and elements. The following code takes advantage of the same generic package STACK to create a stack of 50 records:

```
type EMPLOYEE_REC_TYPE is
  record
  ...
  end record;
EMPLOYEE :  EMPLOYEE_REC_TYPE;
package EMPLOYEE_STACK is new STACK(50,EMPLOYEE_REC_TYPE);
```

Any number of instantiations of the generic package may be active at the same time. That is, we can use REAL_STACK.PUSH(X) to insert a real value into the REAL_STACK and EMPLOYEE_STACK.PUSH(EMPLOYEE) to insert a record in the EMPLOYEE stack.

Separate Compilation

The ability to separately compile modules further increases the benefit of data abstraction. In particular it can reduce costs of program modification [LEBL80]. LeBlanc points out three relevant aspects of compilation cost saving. First, recompiling only affected modules instead of the entire program saves programmer time and computer time. Second, separate compilation facilitates providing a library of reusable code. Third, perhaps the most important cost of "non-separate" compilation is indirect. Using a language which does not facilitate separate compilation conditions the programmer to think of a program as one big chunk rather than a collection of separate units which work in harmony during execution. This tends to foster the bad habit, during modification and new design, of writing a single program to solve the whole problem.

The CALL statement was added to COBOL in 1974 to provide separate

compilation of modules and to achieve the effect of a subroutine with parameters. Each separately compiled module must be a complete program with all 4 divisions. Data items which are to be transferred between two compilation units are listed in a USING clause in each unit. Parameters are passed by reference; that is, any reference to a parameter name in the called module actually accesses the corresponding parameter name in the calling module. Recursive calls to a module are not permitted. Here is a typical example, from [TRIA75]:

Calling Module	Called Module
IDENTIFICATION DIVISION.	IDENTIFICATION DIVISION
PROGRAM_ID. CUST.	PROGRAM_ID. CHECKDIG.
...	...
DATA DIVISION.	DATA DIVISION.
...	...
WORKING_STORAGE SECTION.	WORKING_STORAGE SECTION.
77 CUST_NO PIC X(10).	...
77 CHARS PIC 99.	
	LINKAGE SECTION.
	77 CHARS PIC 99.
	77 CODE_NO PIC 9(10).
PROCEDURE DIVISION.	PROCEDURE DIVISION.
...	
CALL "CHECKDIG"	USING CODE_NO CHARS.
USING CUST_NO CHARS.	...

In the above example, whenever the data item CODE_NO is used in the called module, the variable CUST_NO in the calling module is accessed.

Ada provides more flexibility. A compilation unit can consist of a subprogram declaration or body, a package specification or body, or a generic declaration. Parameters are passed by value, and they may be specified in procedures as "in" or "out" to indicate the direction of data transfer. Recursive calls to a compilation unit are permitted. Ada separate compilation provides the same degree of type checking across compilation units as within units. Relationships between

units are easily established using WITH clauses. Version control will check obsolescence and facilitate top down development [ICHB80].

The Ada package specification and package body are maintained separately by the library facility, and a program which uses a package depends only on the specification. The body can be revised and recompiled, provided it continues to implement the specification, without changing the specification.

Top down design depends on extensive use of modules and submodules. In Ada one can easily specify the existence of submodules while delaying the details of implementation and the compilation of the submodule until later. Revisiting the stack provides a simple example. A good top down programmer might initially define the overall structure of STACK with the following package body [BARN80] which declares separate subunits for the operations PUSH and POP:

```
package body STACK is
   MAX:constant := 100;
   S:array(1..MAX) of REAL;
   PTR:INTEGER range 0..MAX;
   procedure PUSH(X:REAL) is separate;
   function POP return REAL is separate;
begin
   PTR := 0;
end STACK;
```

The separately compiled procedure for PUSH would look something like this:

```
separate(STACK)
procedure PUSH(X:REAL) is
begin
  PTR := PTR + 1;
  S(PTR) := X;
end PUSH;
```

Program Design Language

Ada abstraction capabilities may make it a good choice for a program design language (PDL), even if the ultimate program is coded in another language. When Ada compilers are available it will seldom be necessary to code in another language. There seems to be growing support for the use of Ada as a PDL. The Air Force has selected Ada as the best "redesign language" to convert a large MIS to run on new hardware [FILI80]. IBM Federal Systems Division intends to use an annotated Ada for a PDL for all new software development, and to eventually use Ada for the programming as well [WAR 80]. IBM likes Ada support for design by stepwise refinement and for separating specification from design. TRW believes Ada will help make software usable in more than one project (the reusable code idea) [HART80]. They are investigating use of Ada as a PDL. The Army Center for Tactical Computer Systems has two ongoing software projects for which Ada will be the PDL and coding language [KERN80].

User Defined Types

A splendid example of the utility of a user defined enumeration type in handling control flow is presented in [ATKI78]. The situation

is a common one. You wish to search an array for a certain item, but you are not certain whether or not the item is in the array. Let me first suggest a COBOL solution:

```
01 TABLE.
   05 TABLE_A OCCURS 100 TIMES,
      PICTURE ...
...
77 END_OF_TABLE PIC 999 VALUE 100.
77 TO_END_OF_TABLE PIC 999.
77 INDEX PIC 999.
...
MOVE 1 TO INDEX.
PERFORM SEARCH_LOOP UNTIL INDEX > END_OF_TABLE
                      OR TABLE_A (INDEX) = ITEM .
...
SEARCH_LOOP.
  ADD 1 TO INDEX.
CHECK_STATE.
  IF INDEX > END_OF_TABLE THEN ...  ITEM ABSENT
  ELSE ...  ITEM FOUND.
```

In this search there are three states of interest:

- I have not yet located the item, but I am still searching.
- I found it.
- I have searched the whole array and the item is not in it.

This observation causes us to think of using an enumeration variable

which can take as values the three states. Notice the declaration and use of the variable SEARCHSTATE in the following Ada solution, a modification of the Pascal program in [ATKI78]:

```
ENDOFTABLE :  constant := 100; -- size of array
type INDEXRANGE is range 1 ..  ENDOFTABLE;
type SEARCHSTATE is (SEARCHING, ITEMFOUND, ITEMABSENT);
...
TABLE :  array(INDEXRANGE) of ITEM;
HERE :  INDEXRANGE;
OUTCOME :  SEARCHSTATE;
...
HERE := 1;
OUTCOME := SEARCHING;
loop
  if TABLE[HERE] = ITEMWANTED then OUTCOME := ITEMFOUND;
  elsif HERE = ENDOFTABLE then OUTCOME := ITEMABSENT;
  else HERE := INDEXRANGE'succ(HERE);
exit when OUTCOME <> SEARCHING;
end loop;
case OUTCOME is
  when ITEMFOUND => ...;
  when ITEMABSENT => ...;
end case;
```

As Atkinson points out, the Ada program is better for several reasons [ATKI78]:

- the purpose of the program is more evident

- the program can easily be extended to handle other than the three simple cases

- after exit from the loop it is easy to determine "what state you're in".

A similar approach can be used in COBOL, because one can use a string variable to simulate an enumeration variable, but user defined enumeration types are a more natural and more efficiently implemented solution.

Reusable Code

Winograd maintains that programming languages are not really adequate to resolve the ongoing software crisis [WIN079]. Evidence of the crisis is software which fails to satisfy user needs and is too costly to develop, maintain, modify and reuse [FISH78]. For though programming languages are logically adequate, programmers still are overcome by the complexity of large systems and are perplexed by the difficulty of maintaining code written by someone else [WIN079]. Winograd believes the basic problem is an outmoded idea of the programmer's job: that one conceives of an algorithm for accomplishing a task and then codes it in the precise statements of a programming language. The purpose of a high level language, in this old fashioned view, is to provide tools for stating control instructions and expressing data structure at a higher level than the underlying machine. This purpose of a HOL is still valid, but it should no more be in the forefront of programmer concern than binary arithmetic.

Today's large scale systems demand a different approach to programming. Winograd cites three changes in the basic nature of programming:

- The primary use of computers has shifted from solution of well defined, mathematical or data processing functions to service in a larger, complex system. The system may be a weapons system or a large military organization with a complicated structure and mission.

- The "building blocks" used to construct systems are more broad than programming language features. They are subsystems, "an integrated collection of data structures, programs and protocols" [WIN079] which represent the solution to a part of a real world problem. [Perhaps Winograd meant to use the word "should" to refer to the practice of engineering new software by linking together existing modules, for though this is widely recognized as desirable it is on the fringe of the state of the art]

- Programmers spend most of their time not in generating new programs, but in integrating, modifying and documenting existing ones. Winograd points out that this last trend results from the first two. The ability to field more complex systems leads to systems that keep growing to suit the demands of the environment [WIN079].

This has certainly been the case at Computer Systems Command in recent years. More than half of the command resources are devoted to system maintenance (including correction of errors and modifications). Even "new" system developments are usually rewrites or extensions of a previously existing system. In such an environment there are many common sub functions among the active systems. Recognizing the great

potential conservation of resources, top level managers have been urging and at times directing an increase in the use of reusable code.

This is a recognition of the recurring functions in new systems or system enhancements which, if precautions are not taken, are designed and coded from scratch every time. A great leap forward in programmer productivity is theoretically possible by taking advantage of existing code whenever possible.

Defining what "reusable code" is and attempting to quantify the extent of its use is a knotty problem for system developers. No clear definition exists, but a recent survey uncovered some common practices and attitudes [SMAR80]. Reusable code is variously interpreted to include standard data divisions, common subroutines (often for read, write or edit functions), utilities provided by system library, and program skeletons. Most system developers recognize the potential cost savings and they support increased use of reusable code. Yet no one believes that USACSC has sufficiently exploited this technique.

There are limitations on taking advantage of the reusable code concept. USACSC develops systems to comply with the functional specifications of the customers. If specifications are not common, then code cannot be common either [SMAR80]. Reusable code, then, depends on "reusable specifications". There is some truth in this assertion, but beneath the surface of customer functional specifications, down in the successive levels of abstractions of an implementation, there is great potential for employing reusable code

which does not necessarily resemble a customer specification. Perhaps this concept can be labelled, "reusable abstractions." The idea is poorly supported by COBOL but nicely done in Ada.

Mechanisms available in USACSC to support reuse of code are the source library system and the COBOL INCLUDE, COPY and CALL features.

The COBOL COPY command allows a programmer to easily retrieve and insert previously defined DATA Division code, or any other code, into his program. This feature somewhat supports the reusable code concept but restricts one to reusing code "as is." This is much less flexible than a macro facility or the Ada package. Problems which reduce the utility of the COPY scheme include:

1. All declarations and data structures are global. If a programmer wants to borrow an existing DATA Division, all or part, and write a new procedure division there is no problem -- assuming he carefully uses the predefined data objects in accordance with their declarations. But in the general case he may want to take advantage of several procedures from different programs. These procedures may have data declarations conflicting with one another or with those in the new program being developed. So a piece of code cannot conveniently be "plugged in" to another program. Hidden side effects may necessitate a mammoth debugging phase.

2. There is no interface specification for procedures. Such a specification, describing the data types of the parameters and all other external aspects of the procedure, is needed by both the compiler and the programmer. The compiler needs it to link up the program segments and to perform type checking. Of course COBOL does

little type checking and the link up is simply in line substitution. The programmer must examine the specification of candidate procedures in order to intelligently shop for code that meets his needs and to determine what he requires to interface with it. Comments in English text could aid in this regard but they lack the necessary formality; too much is open to interpretation.

Though the COBOL COPY command is the most prevalent means in USACSC of invoking reusable code, the CALL facility is potentially more useful, since it does permit parameterization and separate compilation. This feature is not selected very often because COBOL does not provide a convenient library or catalogue, there is no provision for formal description of module specifications, and there is little type checking between separately compiled modules. Ada offers an improvement in this regard.

Though Winograd's aforementioned observations are sound, I believe programming languages will play an important role in system development until the state of the art advances appreciably "beyond programming languages." Ada was designed to be more suitable than previous languages in dealing with the software crisis. Ada abstraction features previously described should help make reusable code a reality.

This is the belief of Ichbiah, principal Ada language designer, who forecasted that one of Ada's most important contributions to software engineering will be its support for reusable software components [ICHB80]. The most significant Ada features in this regard are packages, visibility (naming) rules, separate compilation and

provisions for a library of program units [ICHB80]. Ichbiah hopes that the package specification will come to be regarded as a contract with the package user, and the package body will be the fulfillment of the contract. Barnes believes Ada abstraction facilities will encourage creation of reusable software libraries for all types of applications, not just numerical analysis [BARN80]. Ichbiah even sees the formation of a "package industry" of software components, and believes the time is approaching when software modules will be guaranteed.

The guarantee will be like that of a watchmaker: valid as long as case remains intact and no one has tampered with the contents. If Ada packages (or the equivalent) were not available, on receiving a complaint on a routine the "watchmaker" would have to check the rest of the program and all programs which use it to insure there had been no tampering with variables of the routine. There is no danger that routines external to the Ada package can cause any mischief with its inner workings. But it is quite possible for a customer to look inside the watchcase at the package body and to make assumptions about it which may later be invalid if the body is revised for better efficiency. The best course of action, according to Ichbiah, is not to show the implementation to the user. This is feasible in Ada since the package body may be compiled separate from the specification.

Ada provides the same degree of type checking across compilation units as within a unit, so there is no penalty to pay in this regard. Separately compiled units are easily related logically by WITH clauses. The library facility makes precoded modules conveniently

available and checks on obsolescence during program development as new packages evolve.

Portability

In this discussion, portability refers to the relative ease with which software can be moved from one hardware environment to another. There are two approaches to portability: corrective and predictive [HAGU76]. Corrective methods are necessary to convert an existing program originally written for a particular computer with no portability precautions taken. Predictive methods plan ahead. They generally cost more during development but work best over the life cycle.

A survey of the U. S. Army Computer Systems Command portability problem [BROW76] confirmed that the organization must employ both methods. There exist about 4 million lines of COBOL developed for IBM machines. IBM hardware has monopolized the army MIS market until recent years. Competitive procurements for replacement hardware are causing a proliferation of target machines for centrally developed software. Existing code must be converted to run on multiple architectures, and newly developed systems must be constructed to minimize portability problems.

Even though COBOL is a HOL, barriers to portability introduced by varying machine architectures (word length, instruction set, addressing structure, data representation, compiler implementations, and language extensions) can cause problems when software is moved. Compiler vendors provide COBOL extensions which are handy to programmers

but adversely impact software portability.

The USACSC approach in 1980 is to achieve a predictive strategy by adopting a standard subset of COBOL, selected to facilitate portability. Existing code will be converted on a one time basis to the subset, and all new work will employ the subset. Automated conversion tools will be used both to convert old programs to the subset and to amend subset programs to run on particular target machines. The strategy is sound, but COBOL shortcomings make the job more difficult than it has to be.

COBOL pretends to group all machine dependent characteristics in the Environment Division, a noble attempt to identify changes required to transport a program to a new machine. In reality this information is rather arbitrarily spread among the Environment Division and the File Section of the Data Division [JACK76]. Notice the various places you will find the following machine dependent information [JACK76]:

- Multiple records in each tape block (FD)

- Each tape block is prefixed by a length indicator inaccessible to the user program (FD)

- A file is to be accessed only sequentially (SELECT)

- Designation of the name of the record key item for a file accessed by key (SELECT).

Beyond the sound DISPLAY and COMPUTATIONAL formats for elementary data items, COBOL allows compiler designers to establish special formats for efficient implementation on a particular machine. For example, many IBM compilers implement COMP, COMP-1, -2,-3 and -4. This allows a programmer to take advantage of his knowledge of the

actual machine representation of data. For example, the following trick may be used to save space in a file record by getting an unsigned packed decimal item [JACK76]:

```
03 FIELD-A PICTURE S9(5) COMPUTATIONAL-3.
03 FILLER REDEFINES FIELD-A.
  05 FIELD-B PICTURE XX.
  05 FILLER PICTURE X.
```

FIELD-B will contain the four most significant digits of FIELD-A. Now the programmer can add 1 to FIELD-B by adding 10 to FIELD-A.

Machine dependencies are much more evident in Ada and are accessible in the source language. Every implementation of Ada must include a package *SYSTEM which makes available to the source programmer* certain attributes of the underlying hardware [ADA 80]:

```
package SYSTEM is
  type SYSTEM_NAME is -- implementation defined enumeration type
  NAME :  constant SYSTEM_NAME := -- the name of the system
  STORAGE_UNIT:  constant := --the number of bits per storage unit
  MEMORY_SIZE :  constant := --number of storage units in memory
  MIN_INT :  constant := --smallest integer supported
  MAX_INT :  constant := --largest integer supported
  ...
end SYSTEM;
```

A generous number of other predefined attributes are available , such as:

- SIZE : the number of bits used to implement a type

- MACHINE_ROUNDS : value is true if rounding occurs for arithmetic on a particular type

- MANTISSA : number of machine radix places in the mantissa

- OVERFLOWS : true if the exception NUMERIC_ERROR is raised for computations which exceed the range of real arithmetic.

There are other features which help identify and isolate machine dependencies. Explicit declaration of ranges of numeric variables is helpful, and machine dependent portions of a program can be isolated from the rest of the program in a separate package.

A 1980 software conversion effort of the U. S. Air Force Manpower Personnel Center (AFMPC) is a relevant case study in Ada portability. The AFMPC problem is to convert a large ALGOL based information retrieval system, ATLAS, to run on new hardware. They have decided to convert existing software to a more portable form even before the new hardware is selected, because the time between hardware selection and the due date for the software conversion will be too short to achieve an automatic conversion. Even though no Ada compiler exists in 1980, AFMPC decided that Ada had so many portability advantages over COBOL and FORTRAN that they should translate ATLAS into Ada as a preliminary step in the conversion to new hardware. This is their plan, even though they had to construct an Ada translator for use in the recoding of ATLAS. Further details of this effort can be found in [FILI80].

Exception Handling

Ada was designed for real time systems which must be able to recover from errors and continue operation. Therefore, Ada provides

an exception handling capability. What constitutes an exception is open to interpretation. There are two approaches [ADA 79]. The first is a general view which considers exception handling an ordinary programming solution for dealing with any unusual events -- not just errors. Ada implements the second, more narrow approach: exceptions are synonymous with errors and they mandate a termination of the program unit in which they occur. Exception conditions automatically cause a transfer of control to the user defined exception handler for that condition. The handler achieves whatever recovery actions the programmer has provided. Actions may include restarting the procedure in which the exception occurred, but this would be a new invocation.

Exception handling in COBOL is limited to an ability for certain statements to transfer control to a user defined routine upon occurrence of one of a fixed, restricted set of errors. Usually there is just one error condition which may be handled for each type statement. For example:

```
READ INPUT_FILE AT END GO TO EDIT_ROUTINE.
WRITE MASTER_REC INVALID KEY PERFORM WRITE_ERROR.
COMPUTE PAY = HOURS * RATE ON SIZE ERROR PERFORM
OVERFLOW_ROUTINE.
```

Ada exception conditions are not limited to a predefined set. The programmer can define as many exceptions as are appropriate, and choosing descriptive exception names contributes greatly to program readability and facilitates debugging. Consider the following outline of a procedure used to determine the stockage level of parts having a given part number:

```
procedure  GET_PART_STOCKAGE(PART_NO:PART_NO_TYPE,
                             NUMBER : out INTEGER);


  begin

  ...

  GET_NEXT_PART_RECORD(REC : PART_RECORD);

    --above procedure raises exception END_OF_FILE_ERROR when end
    -- of file is reached before a matching part number is found.

  ...

  exception

    when END_OF_FILE_ERROR => raise INVALID_PART_NO;

      --the END_OF_FILE_ERROR exception is handled here by
      --raising a descriptive exception which will be handled
      --by the routines calling GET_PART_STOCKAGE.
```

A programmer can also handle the five predefined Ada exceptions. These include the too familiar NUMERIC_ERROR (otherwise known as "overflow", "divide check", etc. in other languages) and CONSTRAINT_ERROR, which has the same purpose but pertains to programmer defined range and index constraints rather than underlying hardware limitations.

Is this capability useful in MIS applications? I think so. Permit a digression and consider one sample situation: batch oriented MIS in an Army division in Europe. Like many army sites, the division computer center operates at or near saturation, and abnormal termination of any run may cause serious problems to customers. Most application software is configured such that there are restart points, but several hours processing time can still be lost by a failure between restart points. Lost time usually translates directly to delay in producing output for the customer.

The automated repair part resupply system is one of the most critical to the maintenance of combat readiness. Requisitions from customer organizations are collected, consolidated, and keypunched daily. Typically the daily batch of requisitions is submitted to the computer center at 11:00 P.M. for a reserved block of processing time that begins soon after the input arrives. If the computer and software run smoothly through the night, the output is ready for customer pick up at 7:00 A.M. The most critical part of the output is a punched and interpreted card deck of repair parts release orders used to direct parts to customers. If the output is delayed, then repair parts warehouse workers encounter a work stoppage, and while they are wasting time so are mechanics in customer units who are waiting for parts to repair equipment. This situation illustrates the high cost of a program "abort" in a MIS environment -- real time systems do not have a monopoly on this problem.

The most costly (lengthy to fix) aborts result from exceptions which are not handled by the application software; rather, they are trapped by the operating system or the hardware. Such terminations usually provide few useful clues about what went wrong. Often a cryptic code is the only signal, and when one looks up the description of the code in a system manual one finds it also has no clear relevance to the original problem in the application program.

While local analysts try to troubleshoot, with few clues, the computer center manager receives calls from anxious customers who demand to know when they will get their output. Perhaps enough detail

of the scenario has been presented to establish this point: application programs should handle their own "dirty laundry." As many exceptions as possible should be handled by the program so that the program can continue operation. If it must terminate, the program should at least provide meaningful diagnostic messages.

I believe Ada exception handling provides a convenient, natural system that encourages programmers to fulfill their responsibilities toward exceptions.

Syntax and Structure

Richard Wexelbat [WEXL76], while teaching a course on "The Design of computer Languages and Systems for Human Use," included a whimsical question on his final exams. Students were asked to consider their experience and the insight gained from the course to cite ways that malevolent programming language designers could intentionally make programming difficult and erroneous. Resulting is a catalogue of things which should be avoided in programming language design but are evident in COBOL:

- "Make unnatural restrictions on the use of delimiters" e.g. blanks must be used around arithmetic operators in COBOL. X+Y does not work; X + Y must be used.

- "Do not provide function subprograms "Like most languages, Ada permits a program segment to return a value in the fashion of a mathematical function such as:

```
IF AVG(POOR_STUDENT_GRADES) > (5 + AVG(RICH_STUDENT_GRADES))
   THEN ...
```

In COBOL, in such a case one must use two arguments, one to pass a table of values and one to receive the resulting computation:

```
CALL "AVERAGE" USING POOR_STUDENT_GRADES, POOR_AVG.
CALL "AVERAGE" USING RICH_STUDENT_GRADES, RICH_AVG.
ADD 5 TO RICH_AVG.
IF POOR_AVG IS GREATER THAN RICH_AVG ...
```

- "Place excessive restrictions on arrays and subscripting." Programmers are aggravated by arbitrary restrictions on array bounds. COBOL restricts the lower bound to 1. In Ada, any value may be chosen.

Gannon calls the COBOL IF statement error prone, and the allegation is serious since this is the only selection statement available [GANN78]. Gannon overlooks the GO TO DEPENDING ON, a construct not recommended here but one which indeed provides a multiway branch which can be used for selection. Suppose, for example, that one wanted to add the statement S3 to the ELSE clause of the following IF statement, but the programmer forgot to move the period from after S2 to after S3:

```
IF ...
  THEN
    S1
  ELSE
    S2.
    S3
```

Now S3 will be executed unconditionally, but this fact is not obvious to the programmer. We have here an excellent candidate for a difficult logic error. Even when the position of the period is not mistaken, its meaning can be confusing. The period terminates an

entire statement; it cannot be used to delimit IF's in a nested IF statement.

Ada provides a safer construct with an "end if" to more explicitly terminate an IF statement. Example:

```
if ...
  then
    S1
  else
    S2
end if;
    S3
```

Notice how much easier it is to see that S3 is not part of the else clause.

The structured programming approach encourages using nested IF statements to avoid GO TO's, but psychologists report that the difficulty to understand a sentence increases with the level of "self embedding" [MILL64]. Sime, Green and Guest [SIME73] verified this problem using subjects employing IF statements in simple programs. One commonly occurring situation is the need to select from a number of alternatives based on the value of a variable. A nested IF may be used for this purpose in COBOL. Example:

```
IF TODAY = 'MON'
  THEN PERFORM UPDATE_INITIAL_BALANCE
ELSE IF TODAY = 'FRI'
  THEN PERFORM UPDATE_CLOSING_BALANCE
ELSE IF TODAY = 'TUE' OR TODAY = 'WED' OR TODAY = 'THU'
  THEN PERFORM PRINT_REPORT_TODAY.
```

The above example uses "a nested if that really isn't nested". The statement is rather easy to understand, and its complexity does not

grow with the number of alternatives. Nevertheless, Weinberg, Geller and Plum [WEIN75] advocate use of a more linear construct. The Ada CASE statement seems to provide a more desirable structure, for even though nesting is allowed, it will seldom be necessary because the CASE provides a choice between multiple alternatives. Example:

```
case TODAY is
  when MON => UPDATE_INITIAL_BALANCE;
  when FRI => UPDATE_CLOSING_BALANCE;
  when TUE..THU => PRINT_REPORT(TODAY);
  when SAT..SUN => null;
end case;
```

Ada Programming Support Environment

DoD plans to develop a common support environment for Ada. This is a recognition that the environment and support tools for most existing HOL's have been vendor defined and often leave much to be desired. The Ada Programming Support Environment (APSE) will be an integrated collection of tools "to support development and maintenance of Ada application software throughout its life cycle" [STON80]. Minimum features of the APSE include a text editor, pretty printer, Ada translator, linker, loader, set-use static analyzer, control flow static analyzer, dynamic analyze., and others. The APSE will be extendable to support life cycle management activities: requirements specification, overall system design, programming design, program verification and project management.

Serious efforts already underway give credibility to the APSE vision. Fairley has formulated design considerations for interactive

debugging and testing support environments [FAIR80]. There are several proposed methods for Ada program verification, all worthy candidates for inclusion in APSE. For example, a recently developed formal annotation language, ANNA, could be the basis for a verification tool [KRIE80].

To be fair to COBOL I must acknowledge that a fancy support environment could be designed for COBOL as well. But it appears that Ada is destined, with DoD support, to have a rather powerful, standardized support environment, and it may be too late to try to establish a standard environment for COBOL.

Things Ada Does Not Do Well:

Numeric Editing

COBOL programmers often use a record to format and label numeric data:

```
01 TIME_CARD_DETAIL_LINE.
   03 EMPLOYEE_NAME PICTURE X(20).
   03 EMPLOYEE_TOTAL_EARNINGS PICTURE $$$$.99.
```

If the value '12.34' were stored in EMPLOYEE_TOTAL_EARNINGS the result would be represented as 'b$12.34', where the quotes are added here only as delimiters and the 'b' represents a blank. COBOL has an extensive set of built in features for editing numeric data via record 'templates', and it is not yet clear how Ada can provide a reasonable facsimile. If there is no natural solution that does not mean the

editing cannot be done; it just will not be as easy as it is in COBOL. Many language design experts object to automatic type conversion, but the utility of COBOL numeric editing must not be overlooked. Formatting of MIS output is very important to users and is susceptible to frequent changes. Lack of a convenient Ada solution constitutes a significant shortcoming.

For the time being, an Ada solution is presented for a simple case: fixed point numbers without special editing.

COBOL

```
01 TIME_CARD_DETAIL_LINE.
   03 EMPLOYEE_NAME_PICTURE X(20).
   03 EMPLOYEE_TOTAL_EARNINGS PICTURE 999.99.
```

Ada

```
type TIME_CARD_DETAIL_LINE_TYPE is
  record
    EMPLOYEE_NAME : STRING(1 .. 20);
    EMPLOYEE_TOTAL_EARNINGS : delta .01 range 0.0 .. 999.99;
  end record;
TIME_CARD_DETAIL_LINE : TIME_CARD_DETAIL_LINE_TYPE;
```

Notice the ability to constrain the range of numeric types. This provides the benefit of automatic run time checking for violation of the range -- a potential bug prevention medicine.

Record Input/Output

COBOL records are associated with (bound to) "ordinary" files for input and output and with "sort" files.

The following COBOL example includes an "FD" (File Description) which precedes the record(s) declared to be associated with the file:

```
FD LOAN_MASTER_FILE
   RECORD CONTAINS 33 CHARACTERS
   LABEL RECORDS ARE STANDARD
   DATA RECORD IS LOAN_RECORD.
01 LOAN_RECORD.
   05 ACCOUNT_NUMBER PICTURE 9(6).
   05 NAME           PICTURE X(20).
   05 LOAN_AMOUNT    PICTURE 9(5)V99.
```

The Ada package provides a natural way to associate one or more records with a file. One may declare a record type, record objects of that type, plus READ/WRITE operations on records of that type, all in the same package.

```
package LOAN_MASTER_FILE_PACKAGE is
  type LOAN_RECORD_TYPE is
    record
      ACCOUNT_NUMBER : range 0 .. 999999;
      NAME : STRING(1 .. 20);
      LOAN_AMOUNT : delta .01 range 0.0 .. 99999.99;
    end record;
LOAN_RECORD : LOAN_RECORD_TYPE;
procedure WRITE (RECORD : LOAN_RECORD_TYPE);
procedure READ (RECORD : LOAN_RECORD_TYPE);
end LOAN_MASTER_FILE_PACKAGE;
```

The READ and WRITE procedures contain necessary instructions to read and write records of the specified type to the associated file. The programmer may then use a simple instruction like

```
WRITE(LOAN_RECORD);
```

to write a record to its associated file. Notice that this scheme overloads the WRITE procedure, but there need be no confusion to the programmer or compiler. The compiler will properly distinguish which WRITE procedure is intended by examining the number and type of parameters. Since I assume the programmer will make a separate type declaration for the records associated with each file, the overloading should work as desired and allow the programmer to express his I/O instructions simply and logically. The programmer may define several records, each of which represents a format (data template) for data in the file. This is convenient when various views of the data in the

record are necessary. The programmer has the illusion that all the various records associated with an input file are appropriately filled for each READ of the file. For output the programmer need fill only one of the defined output file records with data. The chosen record is then written to the associated file. The scheme is summarized by the rule "read a file -- write a record".

Called "aliasing", this ability to define multiple record templates for the same data object is a source of confusion to the reader of a program. Many language design experts frown on allowing more than one name for the same data object. Several other COBOL options also result in aliasing: REDEFINES, RENAMES and SAME AREA. RENAMES is to be deleted in the next edition (an improvement) [COBOL 80].

Consider the convenience of COBOL automatic type conversion which makes it easy to print records of any format. Usually programmers associate a "string type" record with the output file such as:

```
01 PRINT_REC.
   03 FILLER PIC X(1).
   03 PRINT_LINE PIC X(132).
```

The first character is reserved for printer control and is often initialized to SPACE for single spacing. A line of output is first transferred to PRINT_LINE with a statement like MOVE SUMMARY_LINE TO PRINT_LINE, and then the line is printed by the statement WRITE PRINT_REC. Notice that there is no type checking involved; a record of any format may be moved to PRINT_LINE and then printed.

The Ada programmer must be aware of type compatibility. An Ada file is constrained to consist of elements all of the same type. The COBOL ability to intermix varying record types in the same file is

conceptually a free or undiscriminated union of types. In order to conveniently associate various record formats with a particular Ada file, the programmer will probably choose the variant record construct. Paying the penalty of declaring a tag field (which designates the variant) one may achieve much the same capability as COBOL: varied record formats under the same umbrella (record name). The major difference is that the tag must be properly set when a record is created and must be checked each time a record is read. The compiler can save space in allocating storage. Normally space is necessary only for the largest variant; all variants can be overlaid in the same area. The tag field reveals which variant is applicable. It is a matter of taste whether this explicit control is better than automatic type conversion, but the modern trend is toward explicit control.

Ada can mimic the "read a file" scheme of COBOL, since after each record is read the tag can be examined to determine which variant exists. Then the appropriate view of the record is known and available. Example:

```
type LINE_FMT is (HEADING,DETAIL,SUMMARY);
type MASTER_REC_TYPE(TAG:LINE_FMT) is
  record
    case TAG is
      when HEADING =>
        TITLE: constant STRING := "MONTHLY STATEMENT";
        DATE: DATE_TYPE;
      when DETAIL =>
        CHECK_DATE: DATE_TYPE;
        CHECK_AMT: AMT_TYPE;
      when SUMMARY =>
        BAL_LABEL: constant STRING := "BALANCE";
        BALANCE : AMT_TYPE;
    end case;
  end record;
  ...
READ(MASTER_FILE); -- file of MASTER_REC_TYPE
case TAG is
  when HEADING =>  ...;
  when DETAIL =>  ...;
  when SUMMARY => ...;
end case;
```

The simulation of the COBOL scheme may be carried further by defining a READ procedure that reads a variant record, checks the tag, and stores the contents in the appropriate record -- one of a collection of records, one for each variant. This approach is wasteful of storage.

Variant records can be difficult because Ada strong typing is rather unforgiving. An attempt to intermix records on an Ada print file causes a tag problem reminiscent of the famous line from Shakespeare, "Out damned spot" (substitute "tag" for "spot"). Ada strong typing prevents disassociating components of the variant record from the tag. For example, suppose that we define a print file of the same type records as MASTER_FILE:

```
package PRINT_OUT
  is new INPUT_OUTPUT(ELEMENT_TYPE => MASTER_REC_TYPE);
```

There seems to be no way to print the records of this file without also printing the value of TAG! Since the tag usually serves only to discriminate between variants, it makes no sense to print it along with the rest of the record.

One possible solution is to employ unchecked type conversion to get a handle on the record minus its tag.

COBOL provides a report writer tool which simplifies formatting printed output. Ada provides only a basic set of I/O primitives, but a similar facility could be established as a user defined Ada package. LeBlanc has developed a text formatter package with many of the commonly desired features [LEB80a]. The main issue is run time efficiency. It remains to be seen whether an Ada package can compete with COBOL's built-in report writer in run time efficiency.

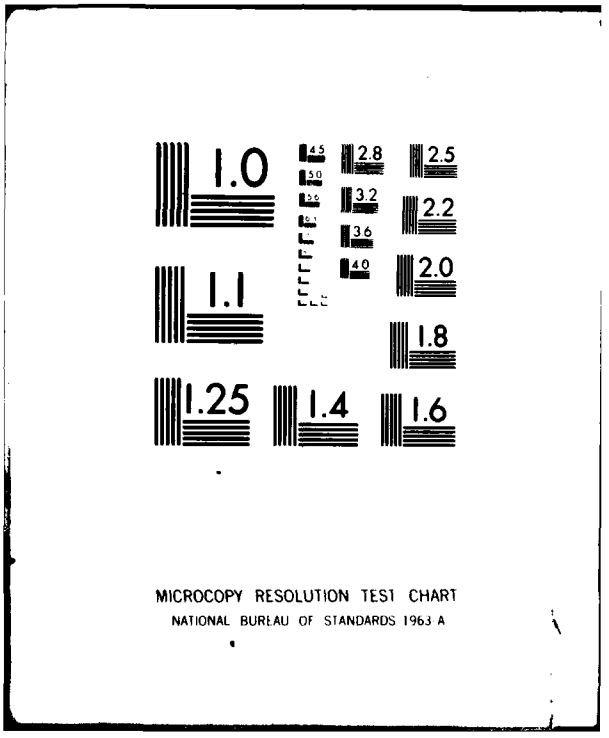
Conclusions

The Ada and MCF programs constitute a revolution in Department of Defense embedded computer technology. As Ada oriented embedded computers begin to dominate the battlefield, military MIS developers will find it increasingly attractive to take advantage of the technology advances associated with "strength in numbers". Adoption of Ada for MIS would reap significant advantages in the "higher level" aspects of software development: management of software development, program design, overall readability and maintainability. The ability to describe module specifications apart from their implementation encourages top down design. Since the package specification can serve as a contract for the programmer of a module, a software development manager can allocate programming work via the specifications. Interfacing the work of a team of programmers is relatively straightforward. Ada strong typing, separate compilation and information hiding help preclude inadvertent damage of the inner workings of one package by another. There should be few surprises when modules are linked together for system testing. Opportunities for reusing code are increased by the Ada facilities for data abstraction.

Since most software development costs are now attributed to the maintenance phase, the use of Ada may be a net improvement over COBOL. Ada programs will tend to be much easier to understand and modify. This should outweigh any additional cost that might be incurred in initial coding, resulting in lower total life cycle cost.

AD-A099 447 ARMY INST FOR RESEARCH IN MANAGEMENT INFORMATION AND --ETC F/G 9/2
ADA - A SUITABLE REPLACEMENT FOR COBOL?(U)
FEB 81 J S DAVIS

UNCLASSIFIED NL


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS 1963 A

Yes, Ada has many advantages, but serious problems must be overcome before Ada is a sound candidate for MIS applications. The Primary challenge confronting prospective users of Ada is the difficulty of retraining COBOL programmers. Ada seems more difficult to learn. At the statement level, Ada is probably less readable than COBOL. Many of its concepts are foreign to even the most accomplished COBOL programmers. At the detailed coding level, after program specifications are completed, COBOL appears to facilitate a faster, more natural solution to problems frequently encountered by programmers of MIS. Convenient COBOL provided features such as numeric editing and record input/output will have to be self constructed Ada tools, at least until a substantial library of reusable "MIS tool kit" packages is developed.

Ada and MCF are not a threat to the COBOL community. Rather they present an opportunity for long term improvement in MIS development. There is no need for haste in deciding what to do. After all, as of 1980 Ada is just a design, but COBOL is a reality with almost two decades of experience. The best tack is a "wait and see" approach, with emphasis on the "see".

References

[ADA 80]
Reference Manual for the Ada Programming Language, Department of Defense, 1980.

[ANSI74]
ANSI X3.23, "American National Standard COBOL," 1974.

[ARMY80]
Memorandum for Army Deputy Chief of Staff for Research, Development and Acquisition, subject: "Standardization of Embedded Computer Resources," 1 July 1980.

[ATKI78]
L.V. Atkinson, "Know the State You Are In," Pascal News #13, December 1978.

[BARN80]
J.G.P. Barnes, "An Overview of Ada," Software Practices and Experiences, Vol. 10, 1980.

[BROW76]
P.J. Brown, "Research on Software Portability," AIRMICS report, 1976.

[BUXT80]
John N. Buxton, "Requirements for Ada Programming Environments -- STONEMAN," February 1980.

[CARR78]
J.C. Carrow, "Structured Programming: From Theory to Practice, "U.S. Army Computer Systems Command paper, 1978.

[COMP80]
Computer Sciences Corporation, "Market Study on U.S. Army Defense Systems," Embedded Computer Systems Market Survey (1979 - 1990)," prepared under contract DAAK 80-79-C-0752, March 1980.

[CLAR73]
B.L. Clark and J.J. Horning, "Reflections on a Language Designed to Write an Operating System," SIGPLAN Notices 8, No. 9, September 1973.

[DAVI80]
J.S. Davis, "MCF: Solution to Computer Proliferation," AIRMICS Report ADA090727, September 1980.

[DEMA80]
Tom DeMarco, "The Ada-Pascal Schism," The Yourdon Report, Vol. 5, No. 4, Aug-Sep 1980.

[DIJK78]
Edsger W. Dijkstra, "DoD-1: The Summing Up," SIGPLAN Notices, Vol. 13, No.7, July 1978.

[DILL77]
George L. Dillon, "Introduction to Contemporary Linguistic Semantics," Prentice-Hall, 1977.

[ENCY76]
Anthony Ralston, Ed., "Encyclopedia of Computer Science," Van Nostrand Reinhold Company, 1976.

[FAIR80]
F.E. Fairley, "Ada Debugging and Testing Support Environments," ACM SIGPLAN Symposium on the Ada Programming Language, Vol. 15, No. 11, November 1980.

[FILI80]
Gary L. Filipski, Donald R. Moore and John E. Newton, "Ada as a Software Transition Tool," ACM SIGPLAN Notices Vol. 15, No. 11, November 1980.

[FISH78]
D.A. Fisher,, "DoD's Common Programming Language Effort," Computer, March 1978.

[GABR80]
Gary Gabriele, Software Science Project Interim Results (AIRMICS), 1980.

[GANN75]
J.D. Gannon and J.J. Horning, "The Impact of Language Design on the Production of Reliable Software,", SIGPLAN Notices, Vol. 10 No. 6.

[GANN76]
J.D. Gannon, "Data Types and Programming Reliability: Some Preliminary Evidence," Proceedings, Symposium on Computer Software Engineering, Polytechnic Institute of New York, April 1976.

[GANN78]
J.D. Gannon, "Characteristic Errors in Programming Languages," ACM 78, Proceedings of the ANnual Conference, 1978.

[GLAS79]
Robert L. Glass, "From Pascal to Pebbleman ... and Beyond," Datamation, July 1979.

[HAGU76]
S.J. Hague and B. Ford, "Portability - Prediction and Correction," Software - Practice and Experience, 6, 1976.

[HART80]
H. Hart of TRW at Redondo Beach, Presentation at ACM SIGPLAN Ada Symposium, Boston, MA, November 1980.

[HOAR73]
C.A.R. Hoare, "Hints on Programming Language Design," Address at SIGACT/SIGPLAN Symposium on Principles of Programming Languages, Boston, October 1973.

[ICHB79]
Jean D. Ichbiah, et. al., "Preliminary Ada Reference Manual," ACM SIGPLAN Notices, Vol. 14, No. 6, June 1979, Part A.

[ICH79B]
Jean D. Ichbiah, et. al., "Rationale for the Design of the Ada Programming Language," ACM SIGPLAN Notices, Vol. 14, No. 6, June 1979, Part B.

[ICHB80]
Jean D. Ichbiah, "Introduction to Ada," Presentation at ACM SIGPLAN Symposium, Boston, MA, 9 December 1980.

[IRON74]
Department of Defense High-Order Language Working Group, "Department of Defense Requirements for High-Order Computer Programming Languages: IRONMAN," Department of Defense, January 1977.

[JACK76]
M.A. Jackson, "COBOL," Software Engineering, Academic Press, June 1976.

[KERN80]
J. Kernan of Center for Tactical Computer Systems, Presentation at ACM SIGPLAN Ada Symposium, Boston, MA, November 1980.

[KRIE80]
B. Krieg-Bruckner and David C. Luckham, "ANNA: Towards a Language for Annotating Ada Programs," ACM SIGPLAN Symposium on the Ada Programming Language, Vol. 15, No. 11, November 1980.

[TRIA75]
J.M. Triance, "The Significance of the 1974 COBOL Standard," The Computer Journal, Vol. 19 No. 4, 1975.

[WAR 80]
D. War of IBM Federal Systems Division, Presentation at ACM SIGPLAN Ada Symposium, Boston, MA, November 1980.

[KREK79]
D. Krekel, "The Design Goals of Ada in Comparison with the Goals of Other Programming Languages," Angewandte Informatik 10, 1979, pp. 425-428.

[LEBL80]
Richard J. Leblanc and Arthur B. Mccabe, "An Introduction to Data Abstraction," School of Information and Computer Science, Georgia Institute of Technology, 1980.

[LEB80a]
Richard J. Leblanc, Ada examples in GIT Ada Course notes, Georgia Institute of Technology, 1980.

[LEIB80]
Edward Leiblein, "Background and Status of MCF," Briefing to Industry, March 25, 1980.

[LEI80B]
Edward Leiblein, et. al., "Development and Procurement Plan for the Military Computer Family," (draft), March 1980.

[MART79]
Edith W. Martin and Edward Leiblein, "MCF Part V, Software for Embedded Computers," Military Electronics, November 1979.

[MITC80]
J.R. Mitchell, "Observations on the Use of Seven Structured Programming Techniques," presented at COMPSAC 1980.

[MILL64]
G.A. Miller and S. Isard, "Free Recall of Self-embedded Sentences," Information and Control 7, 1964.

[MORR73]
J.H. Morris, "Types Are Not Sets," ACM Symposium on the Principles of Programming Languages, October 1973.

[POTT80]
S. Potts of Computer Corporation of America, Presentation at ACM SIGPLAN Ada Symposium, Boston, MA, November 1980.

[SHAW78]
Mary Shaw, et. al., "A Comparison of Programming Languages for Software Engineering," AD AO 53562, April 1978.

[SMAR80]
R. Smart, "Reducing Maintenance Costs by Reusing Code," USACSC paper, 14 November 1979.

[SIME73]
M.E. Sime, T.R.G. Green and D.J. Guest, "Psychological Evaluation of Two Conditional Constructs Used in Computer Languages," International Journal of Man-Machine Studies, 1973.

[WEIN75]
G.M. Weinberg, D.P. Geller, T.W-S Plum, "IF-THEN-ELSE Considered Harmful," SIGPLAN Notices 10, August 1975.

[WEXL76]
R.L. Wexelblat, "Maxims for Malfeasant Designers , or How to Design Languages to Make Programming as Difficult as Possible," Proceedings of the 2d International IEEE Conference on Software Engineering.

[WHIT79]
William A. Whitaker, "Department of Defense Requirements for the Programming Environment for the Common High Order Language - PEBBLEMAN Revised," January 1979.

[WINO79]
T. Winograd, "Beyond Programming Languages," Communications of the ACM, Vol. 22, No. 7, July 1979.

[WIRT74]
Niklaus Wirth, "On the Design of Programming Languages," Information Processing 74, North-Holland Publishing Company, 1974.

[WULF80]
William A. Wulf, "Trends in the Design and Implementation of Programming Languages," Computer, January 1980.


DATE
ILME